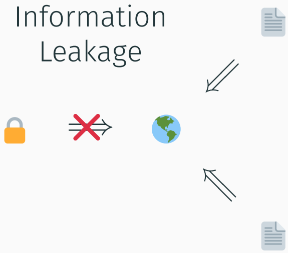
Structural Information Flow: A Fresh Look at Types for Non-Interference

Hemant Gouni with Frank Pfenning & Jonathan Aldrich

MWPLS 2025 and previously OOPSLA 2025

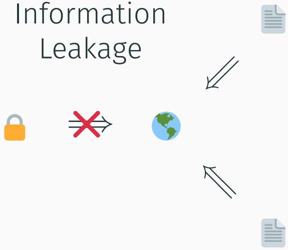
Information Flow Tracks *Dependencies*

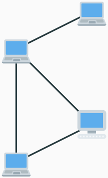
Information Flow Tracks *Dependencies*



Information Flow Tracks *Dependencies*

Information
Leakage

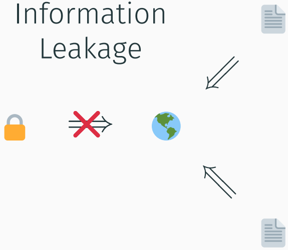




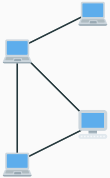
Concurrent
+
Distributed
Systems

Information Flow Tracks *Dependencies*

Information
Leakage



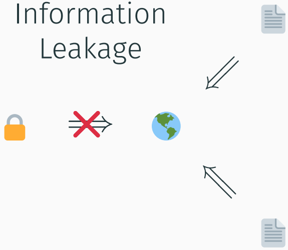
Program Slicing

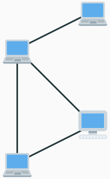


Concurrent
+
Distributed
Systems

Information Flow Tracks *Dependencies*

Information
Leakage





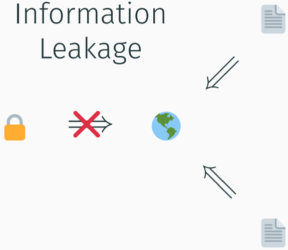
Concurrent
+
Distributed
Systems

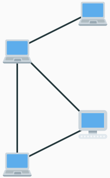
Program Slicing

Build Systems

Information Flow Tracks *Dependencies*

Information
Leakage





Concurrent
+
Distributed
Systems

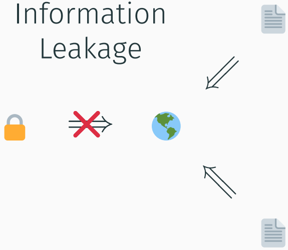
Program Slicing

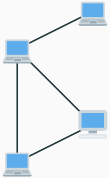
Build Systems

Incremental Computation

Information Flow Tracks *Dependencies*

Information Leakage





Concurrent
+
Distributed Systems

Program Slicing

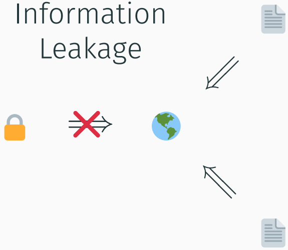
Build Systems

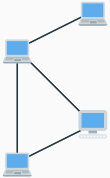
Incremental Computation

Controlling Opaque Definitions

Information Flow Tracks *Dependencies*

Information Leakage





Concurrent + Distributed Systems

Program Slicing 🔪🍖

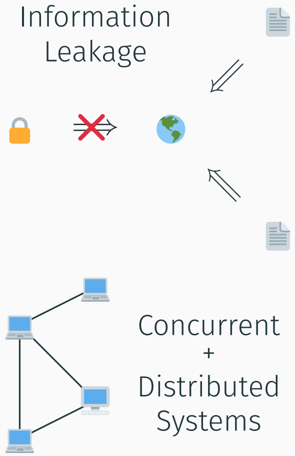
Build Systems ⚙️

Incremental Computation 🤏🔄

Controlling Opaque Definitions 😶‍🌫️

Proof Irrelevance + Type Erasure 📝

Information Flow Tracks *Dependencies*



Program Slicing

Build Systems

Incremental Computation

Controlling Opaque Definitions

Proof Irrelevance + Type Erasure

...and **much** more!!

How to Reinvent Our Approach From Scratch 🧑‍🍳

A Familiar Friend: Parametric Polymorphism

val id : α -> α

val snd : α -> β -> α

val map : (α -> β) -> list α -> list β

A Familiar Friend: Parametric Polymorphism

val id : α -> α

val snd : α -> β -> α

val map : (α -> β) -> list α -> list β

A Familiar Friend: Parametric Polymorphism

```
val id : α -> α

val snd : α -> β -> α

val map : (α -> β) -> list α -> list β
```

A Familiar Friend: Parametric Polymorphism

```
val id : α -> α

val snd : α -> β -> α

val map : (α -> β) -> list α -> list β
```

A Familiar Friend: Parametric Polymorphism

```
val id : α -> α

val snd : α -> β -> α

val map : (α -> β) -> list α -> list β
```

A Familiar Friend: Parametric Polymorphism

```
val id : α -> α

val snd : α -> β -> α

val map : (α -> β) -> list α -> list β
```

What about `incr : int -> int`?

A Familiar Friend: Parametric Polymorphism

val id : α -> α

val snd : α -> β -> α

val map : (α -> β) -> list α -> list β

What about `incr : int -> int`?

Insight 1

Separate **data abstraction** from **information flow**.

Relaxing Parametricity

Insight 1

Separate **data abstraction** from **information flow.**

Relaxing Parametricity

Insight 1

Separate **data abstraction** from **information flow.**

Idea: Tag types with *dependency variables* α

Relaxing Parametricity

Insight 1

Separate **data abstraction** from **information flow.**

Idea: Tag types with *dependency variables* α

Result: `incr :` α `int ->` α `int`

Relaxing Parametricity

Insight 1

Separate **data abstraction** from **information flow.**

Idea: Tag types with *dependency variables* α

Result: `incr :` α `int ->` α `int`

Relaxing Parametricity

Insight 1

Separate **data abstraction** from **information flow.**

Idea: Tag types with *dependency variables* α

Result: `incr` : α `int` `->` α `int`

What about **add** : α `int` `->` β `int` `->` $\boxed{?}$ `int`?

Relaxing Parametricity

Insight 1

Separate **data abstraction** from **information flow.**

Idea: Tag types with *dependency variables* α

Result: `incr` : α `int` `->` α `int`

What about **`add`** : α `int` `->` β `int` `->` $\boxed{?}$ `int`?

Relaxing Parametricity

Insight 1

Separate **data abstraction** from **information flow.**

Idea: Tag types with *dependency variables* α

Result: `incr : ` α `int -> ` α `int`

What about **`add`** `: ` α `int -> ` β `int -> ` $\boxed{?}$ `int`?

Insight 2

Track **sets of dependencies** in types.

Generalizing to Dependency Sets

Insight 1

Separate **data abstraction** from **information flow.**

Insight 2

Track **sets of dependencies** in types.

Generalizing to Dependency Sets

Insight 1

Separate **data abstraction** from **information flow.**

Insight 2

Track **sets of dependencies** in types.

Idea: Generalize each α to a *dependency set* $[\alpha]$

Generalizing to Dependency Sets

Insight 1

Separate **data abstraction** from **information flow.**

Insight 2

Track **sets of dependencies** in types.

Idea: Generalize each α to a *dependency set* [α]

Result: `add : [`α`] int -> [`β`] int -> [`α β`] int`

Generalizing to Dependency Sets

Insight 1

Separate **data abstraction** from **information flow.**

Insight 2

Track **sets of dependencies** in types.

Idea: Generalize each α to a *dependency set* [α]

Result: `add : [`α`] int -> [`β`] int -> [`α β`] int`

Let's work a more interesting example of information flow!

A Password Checker

```
let pass : [pwd] string = "katya"

let check : [α] string -> [α pwd] bool =
    fun attempt -> attempt == pass
```

A Password Checker

```
let pass : [pwd] string = "katya"

let check : [α] string -> [α pwd] bool =
     fun attempt -> attempt == pass
```

Our checker is too conservative!

A Password Checker

```
let pass : [pwd] string = "katya"

let check : [α] string -> [α pwd] bool =
    fun attempt -> attempt == pass
```

Our checker is too conservative!

The conventional solution to this issue is *declassification*...

A Password Checker

```
let pass : [pwd] string = "katya"

let check : [α] string -> [α pwd] bool =
    fun attempt -> declassify(attempt == pass)
```

Our checker is too conservative!

The conventional solution to this issue is *declassification*...

A Password Checker

```
let pass : [pwd] string = "katya"

let check : [α] string -> [α ~~pwd~~] bool =
     fun attempt -> declassify(attempt == pass)
```

Our checker is too conservative!

The conventional solution to this issue is *declassification*...

...which *subverts* the type system.

Non-interference says dependency tracking must be faithful; declassification opposes it.

Non-interference says dependency tracking must be faithful; declassification opposes it.

> *[Non-interference] is too strict to be usable in realistic programs.*
>
> *— Wikipedia (Information Flow)*

Non-interference says dependency tracking must be faithful; declassification opposes it.

> *Noninterference is over-restrictive for programs with intentional information release (average salary, information purchase and password checking programs are flatly rejected by noninterference).*
>
> *— Sabelfeld and Sands 07*

Non-interference says dependency tracking must be faithful; declassification opposes it.

We resolve this conflict: the tools we’ve already introduced suffice for realistic programs.

Declassification for Free 💸

Step 0: Wishful Thinking 💭

```
signature PasswordChecker = sig





end
```

```
open PasswordChecker as pc
```

Step 0: Wishful Thinking 💭

```
signature PasswordChecker = sig
     dependency pwd




end
```

```
open PasswordChecker as pc
```

Step 0: Wishful Thinking 💭

```
signature PasswordChecker = sig
    dependency pwd

    pass : [pwd] string


end
```

```
open PasswordChecker as pc

let _ : [pwd] string = pc.pass ++ "arren"
```

Step 0: Wishful Thinking 💭

```
signature PasswordChecker = sig
     dependency pwd

     pass : [pwd] string
     check : [α] string -> [α] bool

end
```

```
open PasswordChecker as pc

let _ : [pwd] string = pc.pass ++ "arren"
let _ : [ ] bool = pc.check "nemmerle"
```

Step 0: Wishful Thinking 💭

```
signature PasswordChecker = sig
     dependency pwd

     pass : [pwd] string
     check : [α] string -> [α] bool
     encrypt : [pwd α] string -> [α] string
end
```

```
open PasswordChecker as pc

let _ : [pwd] string = pc.pass ++ "arren"
let _ : [ ] bool = pc.check "nemmerle"
let _ : [ ] string = pc.encrypt pc.pass
```

Step 1: Expose Lurking Quantifiers 🥸

```
id  :  α -> α

map :  (α -> β) -> list α -> list β

add :  [α] int -> [β] int -> [α β] int
```

Step 1: Expose Lurking Quantifiers 🥸

```
id  : forall α  . α -> α

map :             (α -> β) -> list α -> list β

add :             [α] int -> [β] int -> [α β] int
```

Step 1: Expose Lurking Quantifiers 🥸

```
id  : forall α   . α -> α

map : forall α β . (α -> β) -> list α -> list β

add :              [α] int -> [β] int -> [α β] int
```

Step 1: Expose Lurking Quantifiers 🥸

```
id  : forall α   . α -> α

map : forall α β . (α -> β) -> list α -> list β

add : forall α β . [α] int -> [β] int -> [α β] int
```

Step 1: Expose Lurking Quantifiers 🥸

```
id  : forall α   . α -> α
map : forall α β . (α -> β) -> list α -> list β
add : forall α β . [α] int -> [β] int -> [α β] int
```

Insight 3

Construct `exists` α from `forall` α + higher-order functions

Step 1: Expose Lurking Quantifiers 🥸

```
id  : forall α   . α -> α

map : forall α β . (α -> β) -> list α -> list β

add : forall α β . [α] int -> [β] int -> [α β] int
```

Insight 3

Construct `exists` α from `forall` α + higher-order functions

Existentials are better known as *modules* or *classes*!*

*(roughly)

Step 2: Dependency Abstraction 🤯

```
signature Queue = sig
  type t

  enqueue : int -> t -> t
  dequeue : t -> t * int
end

structure naive_queue : Queue = struct
  type t = List int

  enqueue x q = Cons(x, q)
  dequeue q = match q with ...
end
```

Step 2: Dependency Abstraction 🤯

```
signature Queue = sig
  type t ←——————— Existentially Quantified!

  enqueue : int -> t -> t
  dequeue : t -> t * int
end

structure naive_queue : Queue = struct
  type t = List int

  enqueue x q = Cons(x, q)
  dequeue q = match q with ...
end
```

Step 2: Dependency Abstraction 🤯

client view

```
signature Queue = sig
  type t

  enqueue : int -> t -> t
  dequeue : t -> t * int
end
```

```
structure naive_queue : Queue = struct
  type t = List int

  enqueue x q = Cons(x, q)
  dequeue q = match q with ...
end
```

Step 2: Dependency Abstraction 🤯

implementation view

```
signature Queue = sig
  type t = List int

  enqueue : int -> List int -> List int
  dequeue : List int -> List int * int
end

structure naive_queue : Queue = struct
  type t = List int

  enqueue x q = Cons(x, q)
  dequeue q = match q with ...
end
```

Step 3: Our New Password Checker 🎉

```
signature PasswordChecker = sig
    dependency pwd

    pass : [pwd] string
    check : [α] string -> [α] bool

    encrypt : [pwd α] string -> [α] string

end
```

Step 3: Our New Password Checker 🎉

```
structure pc : PasswordChecker = struct
    dependency pwd

    pass : [pwd] string
    check : [α] string -> [α] bool

    encrypt : [pwd α] string -> [α] string

end
```

Step 3: Our New Password Checker 🎉

```
structure pc : PasswordChecker = struct
     dependency pwd = [ ]

     pass : [~~pwd~~] string
     check : [α] string -> [α] bool

     encrypt : [~~pwd~~ α] string -> [α] string

end
```

Step 3: Our New Password Checker 🎉

```
structure pc : PasswordChecker = struct
    dependency pwd = [ ]

    pass : [pwd] string = "katya"
    check : [α] string -> [α] bool

    encrypt : [pwd α] string -> [α] string

end
```

Step 3: Our New Password Checker 🎉

```
structure pc : PasswordChecker = struct
     dependency pwd = [ ]

     pass : [~~pwd~~] string = "katya"
     check : [α] string -> [α] bool
          = fun attempt -> attempt == pass
     encrypt : [~~pwd~~ α] string -> [α] string

end
```

Step 3: Our New Password Checker 🎉

```
structure pc : PasswordChecker = struct
     dependency pwd = [ ]

     pass : [~~pwd~~] string = "katya"
     check : [α] string -> [α] bool
          = fun attempt -> attempt == pass
     encrypt : [~~pwd~~ α] string -> [α] string
          = fun str -> gpg str
end
```

Step 3: Our New Password Checker 🎉

```
structure pc : PasswordChecker = struct
    dependency pwd = [ ]

    pass : [pwd] string = "katya"
    check : [α] string -> [α] bool
        = fun attempt -> attempt == pass
    encrypt : [pwd α] string -> [α] string
        = fun str -> gpg str
end
```

Takeaway: Full-strength non-interference and practical programming are perfectly aligned.

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

Takeaway: Full-strength non-interference and practical programming are perfectly aligned.

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

POPL 2026:

Security Reasoning via Substructural Dependency Tracking

HEMANT GOUNI, Carnegie Mellon University, USA
FRANK PFENNING, Carnegie Mellon University, USA
JONATHAN ALDRICH, Carnegie Mellon University, USA

Substructural type systems provide the ability to speak about *resources*. By enforcing usage restrictions on *inputs* to computations they allow programmers to reify limited system units—such as memory—in types. We demonstrate a new form of resource reasoning founded on constraining *outputs* and explore its utility