Security Reasoning via Substructural Dependency Tracking

Hemant Gouni (with Frank Pfenning & Jonathan Aldrich)

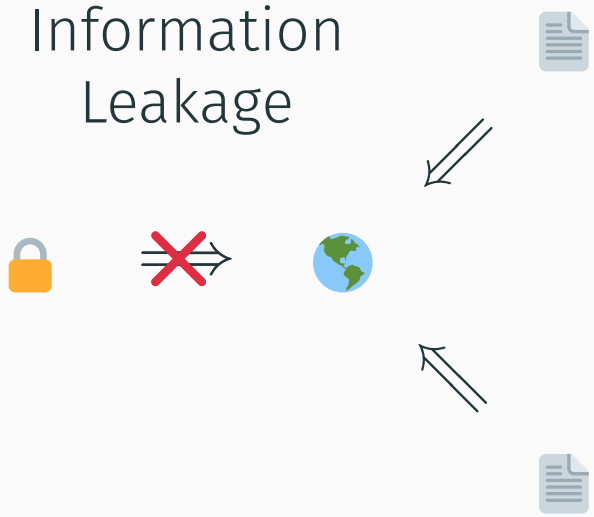
January 15, 2026

Security Reasoning via Substructural Dependency Tracking

Hemant Gouni (with Frank Pfenning & Jonathan Aldrich)

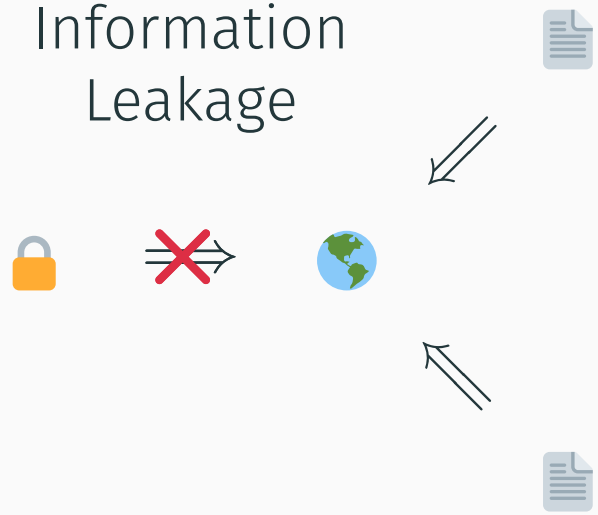
January 15, 2026

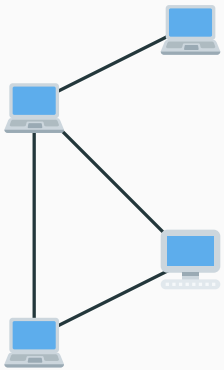
Static Dependency Tracking



Static Dependency Tracking

Information Leakage

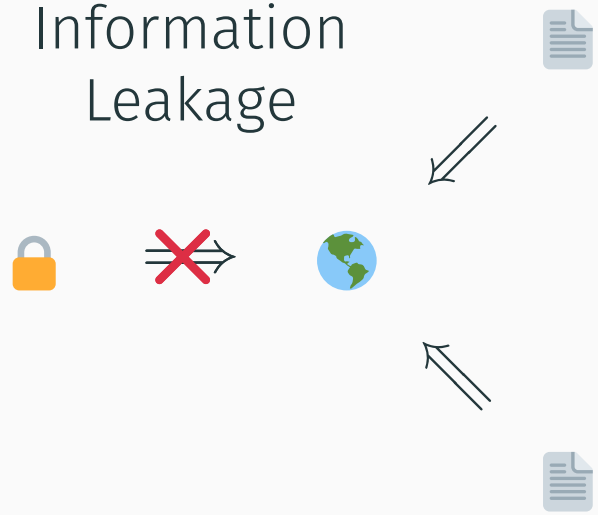




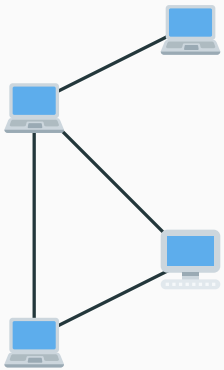
Concurrent
+
Distributed Systems

Static Dependency Tracking

Information Leakage



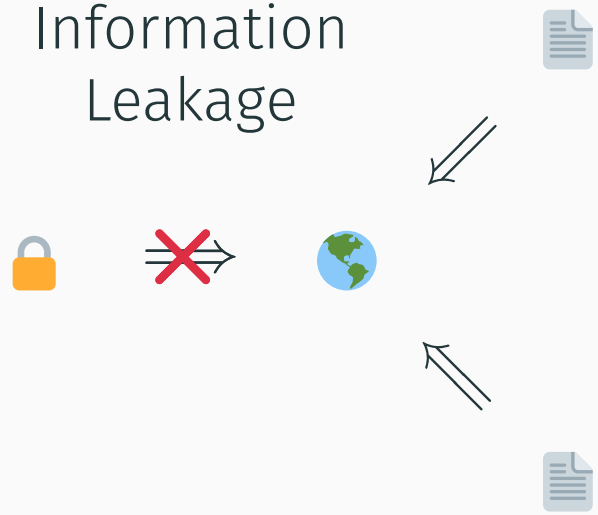
Program Slicing 📣



Concurrent
+
Distributed Systems

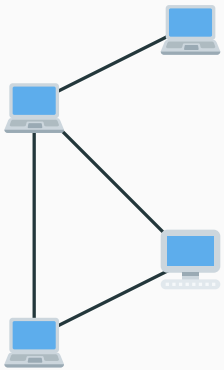
Static Dependency Tracking

Information Leakage



Program Slicing

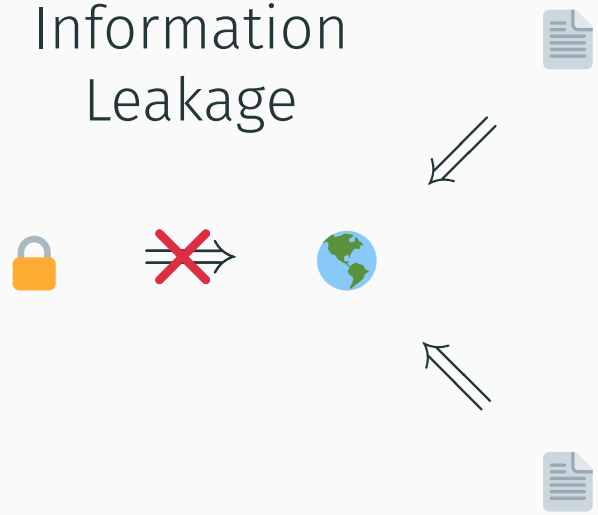
Build Systems

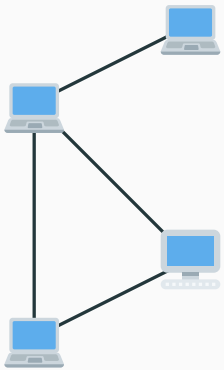


Concurrent
+
Distributed
Systems

Static Dependency Tracking

Information Leakage





Concurrent
+
Distributed Systems

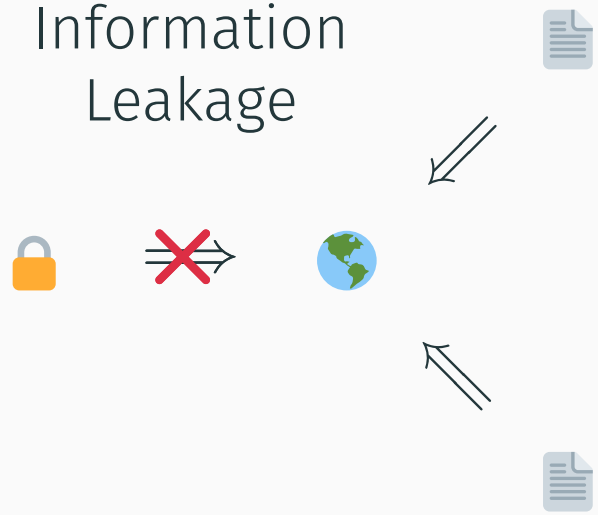
Program Slicing

Build Systems

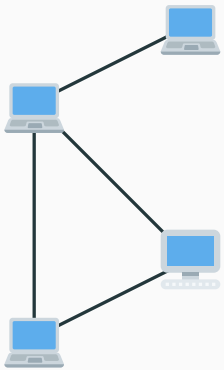
Reactive Programming

Static Dependency Tracking

Information Leakage



Concurrent + Distributed Systems



Program Slicing

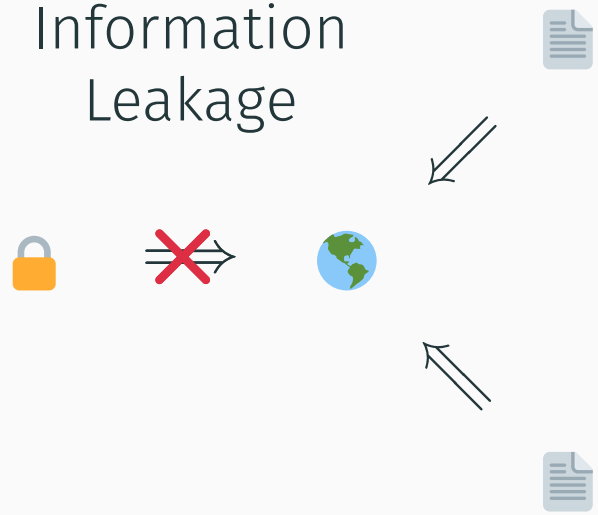
Build Systems

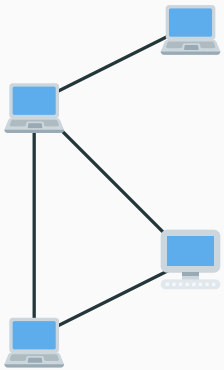
Reactive Programming

Dependent how often?

Static Dependency Tracking

Information Leakage





Concurrent
+
Distributed Systems

Program Slicing

Build Systems

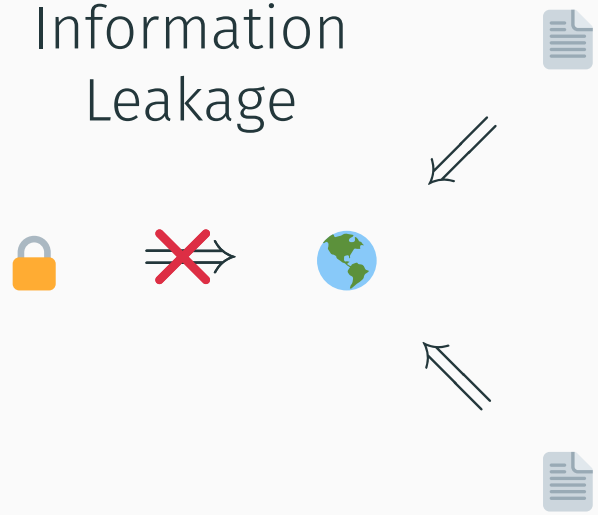
Reactive Programming

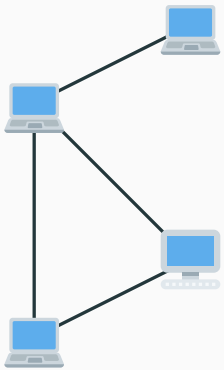
Dependent how often?

Dependency guaranteed?

Static Dependency Tracking

Information Leakage





Concurrent
+
Distributed Systems

Program Slicing

Build Systems

Reactive Programming

Dependent how often?

Dependency guaranteed?

What dependency order?

Reinventing Our Approach 👨‍🍳

Reinventing Our Approach 👨‍🍳

⇓

Substructural

Dependency Tracking

Affine Types in Rust

```
let x: String = format!("hello");
```

Affine Types in Rust

```
let x: String = format!("hello");

func1(x);
```

Affine Types in Rust

```
let x: String = format!("hello");

func1(x);
func2(x);
```

Affine Types in Rust

```
let x: String = format!("hello");

func1(x);  ⟵ x moved here
func2(x);  ⟵ x used here after move ⚠
```

Affine Types in Rust

```
let x: String = format!("hello");

func1(x);  ⟵ x moved here
func2(x);  ⟵ x used here after move ⚠
```

Data in Rust is *ephemeral*: it can only be consumed a **single** time!

Affine Types in Rust

```
let x: String = format!("hello");

func1(x);  ⟵ x moved here
func2(x);  ⟵ x used here after move ⚠
```

Data in Rust is *ephemeral*: it can only be consumed a **single** time! Ephemerality leads to resource reasoning.

Bounded Affine Types

Generalize from **single** to **finitely-bounded** consumption.

Bounded Affine Types

Generalize from **single** to **finitely-bounded** consumption.

```
val f : nat {⛽ ⛽ ⛽} -> nat
```

Bounded Affine Types

Generalize from **single** to **finitely-bounded** consumption.

```
val f : nat {⛽ ⛽ ⛽} -> nat
```

```
let f x = x * x
```

Bounded Affine Types

Generalize from **single** to **finitely-bounded** consumption.

```
val f : nat {⛽ ⛽ ⛽} -> nat

let f x = x * x
let f x = x * x * x
```

Bounded Affine Types

Generalize from **single** to **finitely-bounded** consumption.

```
val f : nat {⛽ ⛽ ⛽} -> nat

let f x = x * x
let f x = x * x * x
let f x = x * x * x * x  ⟵ out of fuel for x ⚠
```

Bounded Affine Types

Generalize from **single** to **finitely-bounded** consumption.

```
val f : nat {🔥 🔥 🔥} -> nat
```

```
let f x = x * x
let f x = x * x * x
let f x = x * x * x * x  ⟵ out of fuel for x ⚠
```

Observe that 🔥 on the **input** dictates the structure of `f`.

Our Approach: Reverse! ↩️

Use output—rather than input—restrictions to create resources.

Our Approach: Reverse! ↩️

Use output—rather than input—restrictions to create resources.

```
val g : [🪙] nat -> [🪙 🪙 🪙] nat

let g x = x * x
let g x = x * x * x
let g x = x * x * x * x  ⟵ 🪙 🪙 🪙 🪙 ⚠️
```

Our Approach: Reverse! ↩️

Use output—rather than input—restrictions to create resources.

```
val g : [🪙] nat -> [🪙 🪙 🪙] nat
```

```
let g x = !x * !x
let g x = !x * !x * !x
let g x = !x * !x * !x * !x  ⟵ 🪙 🪙 🪙 🪙 ⚠
```

The resource count increases when `x` is *run*, not merely *used*.

Our Approach: Reverse! ↩️

Use output—rather than input—restrictions to create resources.

```
val g : [🪙] nat -> [🪙 🪙 🪙] nat
          thunk
let g x = !x * !x
let g x = !x * !x * !x
let g x = !x * !x * !x * !x   ⟵ 🪙 🪙 🪙 🪙 ⚠
```

The resource count increases when `x` is *run*, not merely *used*.

Our Approach: Reverse! ↩️

Use output—rather than input—restrictions to create resources.

```
val g : [🪙] nat -> [🪙 🪙 🪙] nat
        ^^^^^^^
          thunk
let g x = !x * !x
let g x = !x * !x * !x
let g x = !x * !x * !x * !x   ⟵ 🪙 🪙 🪙 🪙 ⚠
          ^^^^^^^^^^^^^^^^^
              force
```

The resource count increases when `x` is *run*, not merely *used*.

Our Approach: Reverse! ↩️

Use output—rather than input—restrictions to create resources.

```
val g : [🪙] nat -> [🪙 🪙 🪙] nat

let g x = !x * !x
let g x = !x * !x * !x
let g x = !x * !x * !x * !x  ⟵ 🪙 🪙 🪙 🪙 ⚠
let g x = bind y to !x in y * y * y * y
```

The resource count increases when `x` is *run*, not merely *used*.

Our Approach: Reverse! ↩️

Use output—rather than input—restrictions to create resources.

```
val g : [🪙] nat -> [🪙 🪙 🪙] nat

let g x = !x * !x
let g x = !x * !x * !x
let g x = !x * !x * !x * !x   ⟵ 🪙 🪙 🪙 🪙 ⚠
let g x = bind y to !x in y * y * y * y
                    ⇒ 🪙
```

The resource count increases when `x` is *run*, not merely *used*.

Our Approach: Reverse! ↩️

Use output—rather than input—restrictions to create resources.

```
val g : [🪙] nat -> [🪙 🪙 🪙] nat

let g x = !x * !x
let g x = !x * !x * !x
let g x = !x * !x * !x * !x   ⟵ 🪙 🪙 🪙 🪙 ⚠
let g x = bind y to !x in y * y * y * y
                ⇒ 🪙          : nat
```

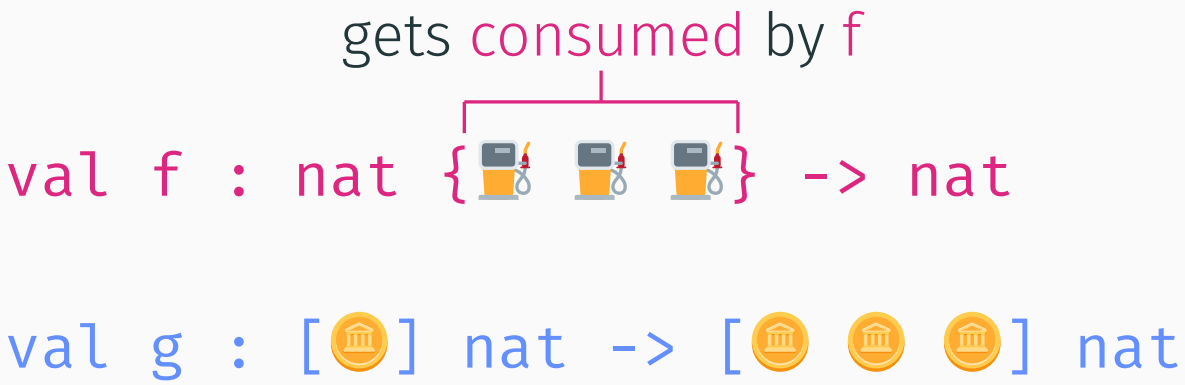
The resource count increases when `x` is *run*, not merely *used*.

Aside: Why the focus on computations?

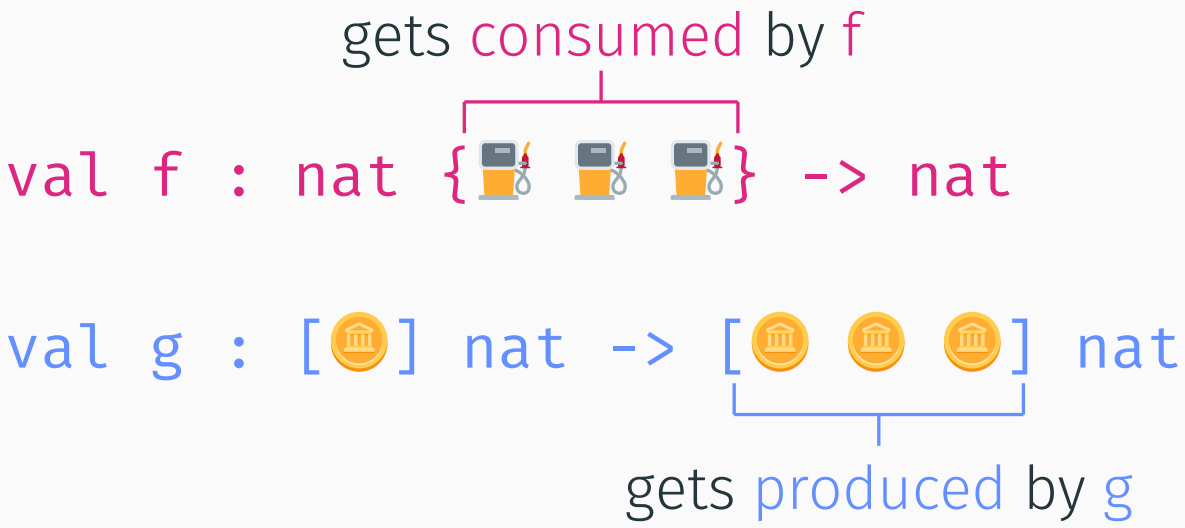
```
val f : nat {🍻 🍻 🍻} -> nat

val g : [🪙] nat -> [🪙 🪙 🪙] nat
```

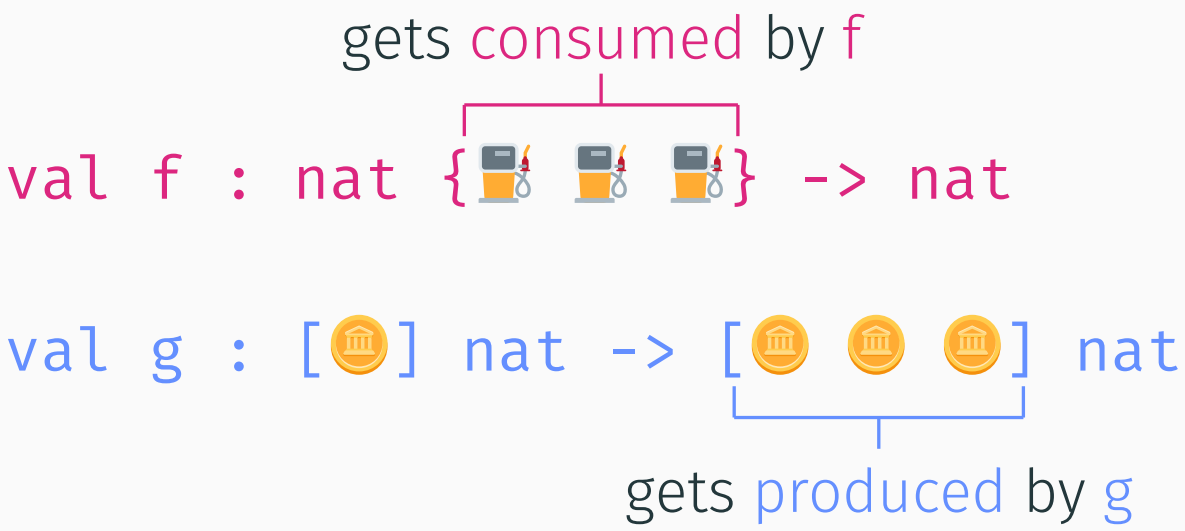
Aside: Why the focus on computations?



Aside: Why the focus on computations?



Aside: Why the focus on computations?



Justification: Consumption vs Production

Coeffects regard consumption via variable use.

Effects regard production via running computations.

Examples

Example: Quantity-Sensitive Leakage

```
module PasswordChecker : sig


end
```

Example: Quantity-Sensitive Leakage

```
module PasswordChecker : sig
      affine resource 🔐

end
```

Example: Quantity-Sensitive Leakage

```
module PasswordChecker : sig
      affine resource 🔐
      val check : string -> [🔐] bool
end
```

Example: Quantity-Sensitive Leakage

```
module PasswordChecker : sig
      affine resource 🔐
      val check : string -> [🔐] bool
end
```

```
open PasswordChecker as pc
```

Example: Quantity-Sensitive Leakage

```
module PasswordChecker : sig
      affine resource 🔐
      val check : string -> [🔐] bool
end
```

```
open PasswordChecker as pc

let _ : [pc.🔐] bool = pc.check "faxe"
```

Example: Quantity-Sensitive Leakage

```
module PasswordChecker : sig
      affine resource 🔐
      val check : string -> [🔐] bool
end
```

```
open PasswordChecker as pc

let _ : [pc.🔐] bool = pc.check "faxe"
let _ : [pc.🔐 pc.🔐] bool =
      pc.check "faxe" && pc.check "tibe"
```

Example: Quantity-Sensitive Leakage

```
module PasswordChecker : sig
      affine resource 🔐
      val check : string -> [🔐] bool
end
```

```
open PasswordChecker as pc

let _ : [pc.🔐] bool = pc.check "faxe"
let _ : [pc.🔐 pc.🔐] bool =
      pc.check "faxe" && pc.check "tibe"
let _ : [pc.🔐] bool =
      pc.check "faxe" && pc.check "tibe"
```

Example: Quantity-Sensitive Leakage

```
module PasswordChecker : sig
      affine resource 🔐
      val check : string -> [🔐] bool
end
```

```
open PasswordChecker as pc

let _ : [pc.🔐] bool = pc.check "faxe"
let _ : [pc.🔐 pc.🔐] bool =
      pc.check "faxe" && pc.check "tibe"
let _ : [pc.🔐] bool =
      pc.check "faxe" && pc.check "tibe" ⚠
```

Example: Quantity-Sensitive Leakage

```
module PasswordChecker : sig
    affine resource 🔐
    val check : string -> [🔐] bool
end

open PasswordChecker as pc

let _ : [pc.🔑] bool = pc.check "faxe"
let _ : [pc.🔐 pc.🔐] bool =
    pc.check "faxe" && pc.check "tibe"
let _ : [pc.🔐] bool =
    pc.check "faxe" && pc.check "tibe" ⚠
```

Example: Quantity-Sensitive Leakage

```
module PasswordChecker : sig
    affine resource 🔐
    val check : string -> [🔐] bool
end

.....................................................................

open PasswordChecker as pc

let _ : [pc.🔑 pc.🔑] bool = pc.check "faxe"
let _ : [pc.🔐 pc.🔐] bool =
    pc.check "faxe" && pc.check "tibe"
let _ : [pc.🔐] bool =
    pc.check "faxe" && pc.check "tibe" ⚠
```

Example: Capabilities

Example: Capabilities

Example: Capabilities


```
module Authorize : sig



end
```

Example: Capabilities


```
module Authorize : sig
      strict resource 💎


end
```

Example: Capabilities


```
module Authorize : sig
      strict resource 💎
      val authenticate : string ->
            unit + [💎] unit
end
```

Example: Capabilities


```
module Authorize : sig
      strict resource 💎
      val authenticate : string ->
            unit + [💎] unit
end
```

```
let secured : [💎] int -> ...
```

Example: Capabilities


```
module Authorize : sig
      strict resource 💎
      val authenticate : string ->
            unit + [💎] unit
end
```

```
let secured : [💎] int -> ...

case authenticate "argaven" with
| Left _ -> ...
| Right (tok : [💎] unit) -> secured (4 <~ tok)
```

Example: Protocols, or `seccomp`

Example: Protocols, or `seccomp`


```
module DropProto : sig




end
```

Example: Protocols, or `seccomp`


```
module DropProto : sig
     immobile resource drp, hi



end
```

Example: Protocols, or `seccomp`


```
module DropProto : sig
     (immobile) resource drp, hi




end
```

Example: Protocols, or `seccomp`


```
module DropProto : sig
     (immobile) resource drp, hi
     structural resource lo



end
```

Example: Protocols, or `seccomp`


```
module DropProto : sig
      (immobile) resource drp, hi
      structural resource lo
      val drop : [drp] unit
      val hi : [hi] unit
      val lo : [lo] unit
end
```

Example: Protocols, or `seccomp`


```
module DropProto : sig
      (immobile) resource drp, hi
      structural resource lo
      val drop : [drp] unit
      val hi : [hi] unit
      val lo : [lo] unit
end
```

```
let _ : [hi drp lo] … = !hi; !lo; !drop; !lo
```

Example: Protocols, or `seccomp`


```
module DropProto : sig
      (immobile) resource drp, hi
      structural resource lo
      val drop : [drp] unit
      val hi : [hi] unit
      val lo : [lo] unit
end
```

```
let _ : [hi drp lo] … = !hi; !lo; !drop; !lo
let _ : [hi drp lo] … = !lo; !hi; !lo; !hi
```

Example: Protocols, or `seccomp`


```
module DropProto : sig
      (immobile) resource drp, hi
      structural resource lo
      val drop : [drp] unit
      val hi : [hi] unit
      val lo : [lo] unit
end
```

```
let _ : [hi drp lo] … = !hi; !lo; !drop; !lo
let _ : [hi drp lo] … = !lo; !hi; !lo; !hi
let _ : [hi drp lo] … = !hi; !drop; !lo;
```

Example: Protocols, or `seccomp`


```
module DropProto : sig
      (immobile) resource drp, hi
      structural resource lo
      val drop : [drp] unit
      val hi : [hi] unit
      val lo : [lo] unit
end
```

```
let _ : [hi drp lo] … = !hi; !lo; !drop; !lo
let _ : [hi drp lo] … = !lo; !hi; !lo; !hi
let _ : [hi drp lo] … = !hi; !drop; !lo; !hi ⚠
```

Versus Conventional Resource Reasoning

The *production* perspective naturally characterizes a different range of resources than the *consumption* one:

Versus Conventional Resource Reasoning

The *production* perspective naturally characterizes a different range of resources than the *consumption* one:

- 🔐 Quantity-Sensitive Leakage

Versus Conventional Resource Reasoning

The *production* perspective naturally characterizes a different range of resources than the *consumption* one:

- 🔐 Quantity-Sensitive Leakage
- 💎 Authorization via Capabilities

Versus Conventional Resource Reasoning

The *production* perspective naturally characterizes a different range of resources than the *consumption* one:

- 🔐 Quantity-Sensitive Leakage
- 💎 Authorization via Capabilities

`drp` `seccomp`-style sandboxing

Versus Conventional Resource Reasoning

The *production* perspective naturally characterizes a different range of resources than the *consumption* one:

- 🔐 Quantity-Sensitive Leakage
- 💎 Authorization via Capabilities

`drp` `seccomp`-style sandboxing

More examples in the paper!

🛒 3-in-1: {Capability, Quantity, Protocol} Safety

Substructurality via Subsumption

- Weakening: $[\ldots] \longmapsto [\ldots$  $]$

Substructurality via Subsumption

- ~~Weakening~~: [...]  [... 💎]

Substructurality via Subsumption

- Weakening: $[\ldots] \sqsubseteq [\ldots$ 💎$]$

Substructurality via Subsumption

- ~~Weakening~~: $[\ldots] \not\sqsubseteq [\ldots$ 💎$]$

Substructurality via Subsumption

- ~~Weakening~~: `[...]` $\not\sqsubseteq$ `[... 💎]` $\Longrightarrow$ `strict`

Substructurality via Subsumption

- ~~Weakening~~: [...] $\not\sqsubseteq$ [... 💎] $\Longrightarrow$ `strict`
- Contraction: [🔐 🔐] $\sqsubseteq$ [🔐]

Substructurality via Subsumption

- ~~Weakening~~: [...] $\not\sqsubseteq$ [... 💎] $\Longrightarrow$ `strict`
- ~~Contraction~~: [🔐 🔐] $\not\sqsubseteq$ [🔐]

Substructurality via Subsumption

- ~~Weakening~~: [...] $\not\sqsubseteq$ [... 💎] $\Longrightarrow$ `strict`
- ~~Contraction~~: [🔐 🔐] $\not\sqsubseteq$ [🔐] $\Longrightarrow$ `affine`

Substructurality via Subsumption

- ~~Weakening~~: `[...]` $\not\sqsubseteq$ `[... 💎]` $\Longrightarrow$ `strict`
- ~~Contraction~~: `[🔐 🔐]` $\not\sqsubseteq$ `[🔐]` $\Longrightarrow$ `affine`
- Exchange: `[hi drp]` $\sqsubseteq$ `[drp hi]`

Substructurality via Subsumption

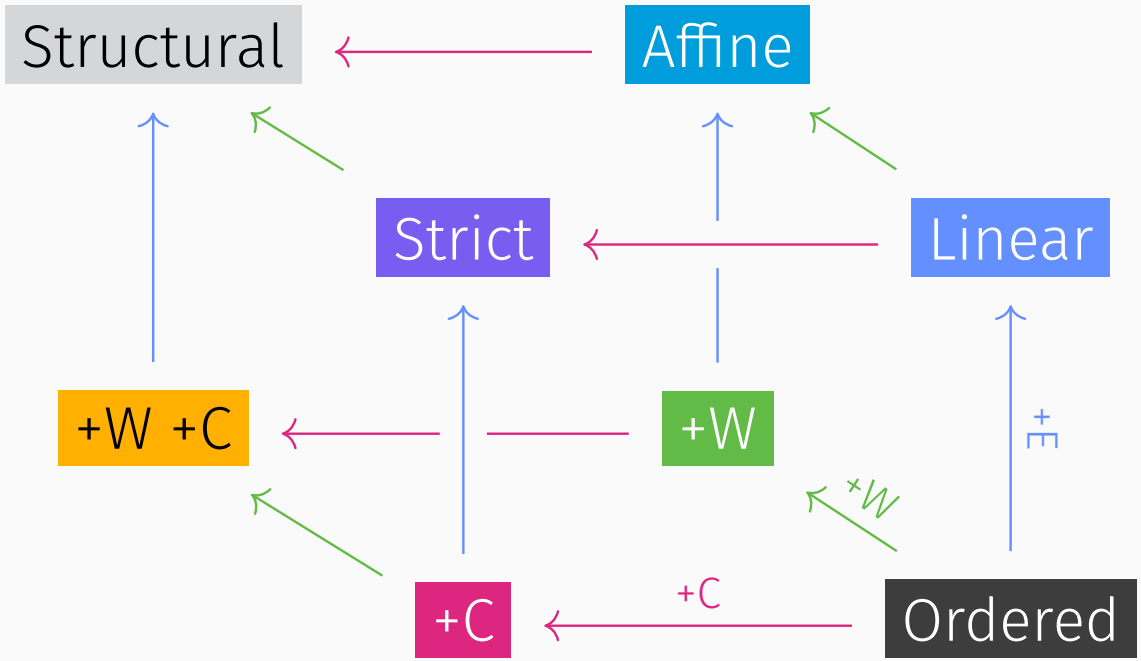
- ~~Weakening~~: `[...]` $\not\sqsubseteq$ `[... 💎]` $\Longrightarrow$ `strict`
- ~~Contraction~~: `[🔐 🔐]` $\not\sqsubseteq$ `[🔐]` $\Longrightarrow$ `affine`
- ~~Exchange~~: `[hi drp]` $\not\sqsubseteq$ `[drp hi]`

Substructurality via Subsumption

- ~~Weakening~~: `[...]` $\not\sqsubseteq$ `[... 💎]` $\Longrightarrow$ `strict`
- ~~Contraction~~: `[🔐 🔐]` $\not\sqsubseteq$ `[🔐]` $\Longrightarrow$ `affine`
- ~~Exchange~~: `[hi drp]` $\not\sqsubseteq$ `[drp hi]` $\Longrightarrow$ `immobile`

Substructurality via Subsumption

- ~~Weakening~~: `[...]` $\not\subseteq$ `[... 💎]` $\implies$ `strict`
- ~~Contraction~~: `[🔐 🔐]` $\not\subseteq$ `[🔐]` $\implies$ `affine`
- ~~Exchange~~: `[hi drp]` $\not\subseteq$ `[drp hi]` $\implies$ `immobile`



Soundness Theorem

If e `: [a1 a2 ...] A` then `!`$e \longmapsto^* v$ producing resources `[b1 b2 ...]` and `[b1 b2 ...]` $\sqsubseteq$ `[a1 a2 ...]`.

Soundness Theorem

well-typed under
expected resources

If e `: [a1 a2 ...] A` then `!`$e \longmapsto^* v$ producing resources `[b1 b2 ...]` and `[b1 b2 ...]` $\sqsubseteq$ `[a1 a2 ...]`.

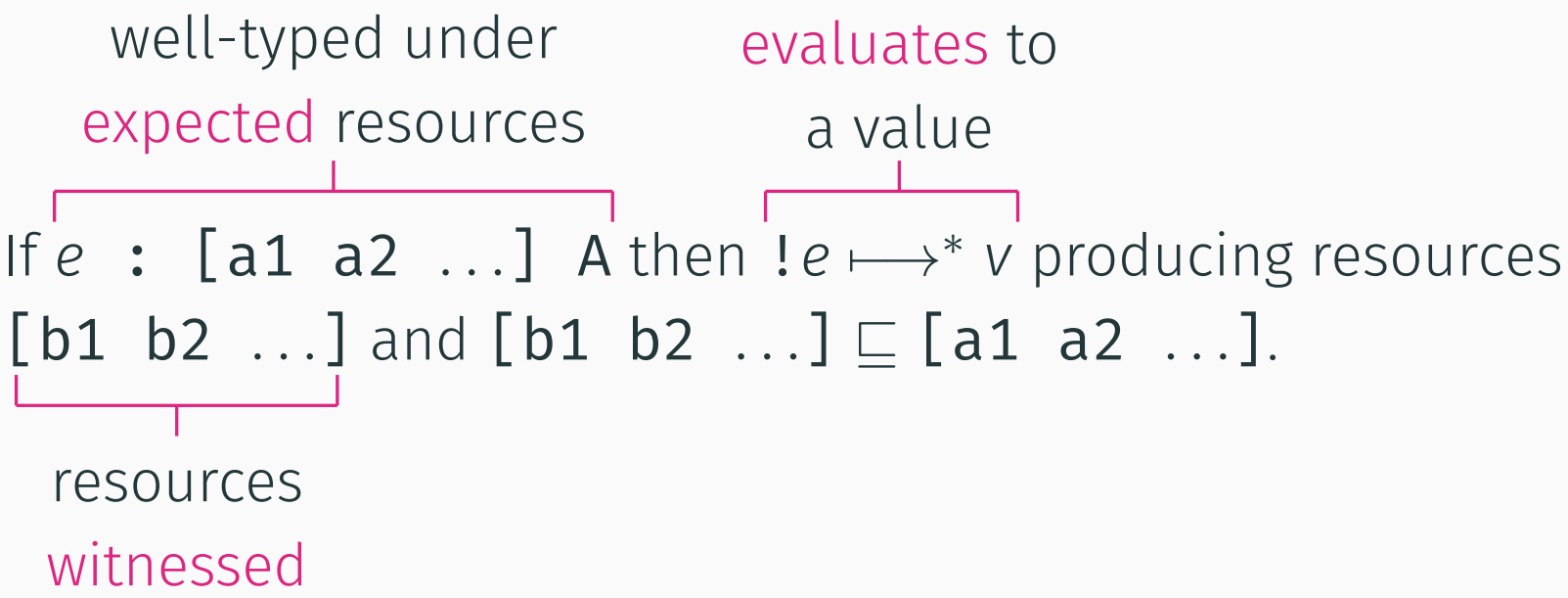
Soundness Theorem

well-typed under expected resources

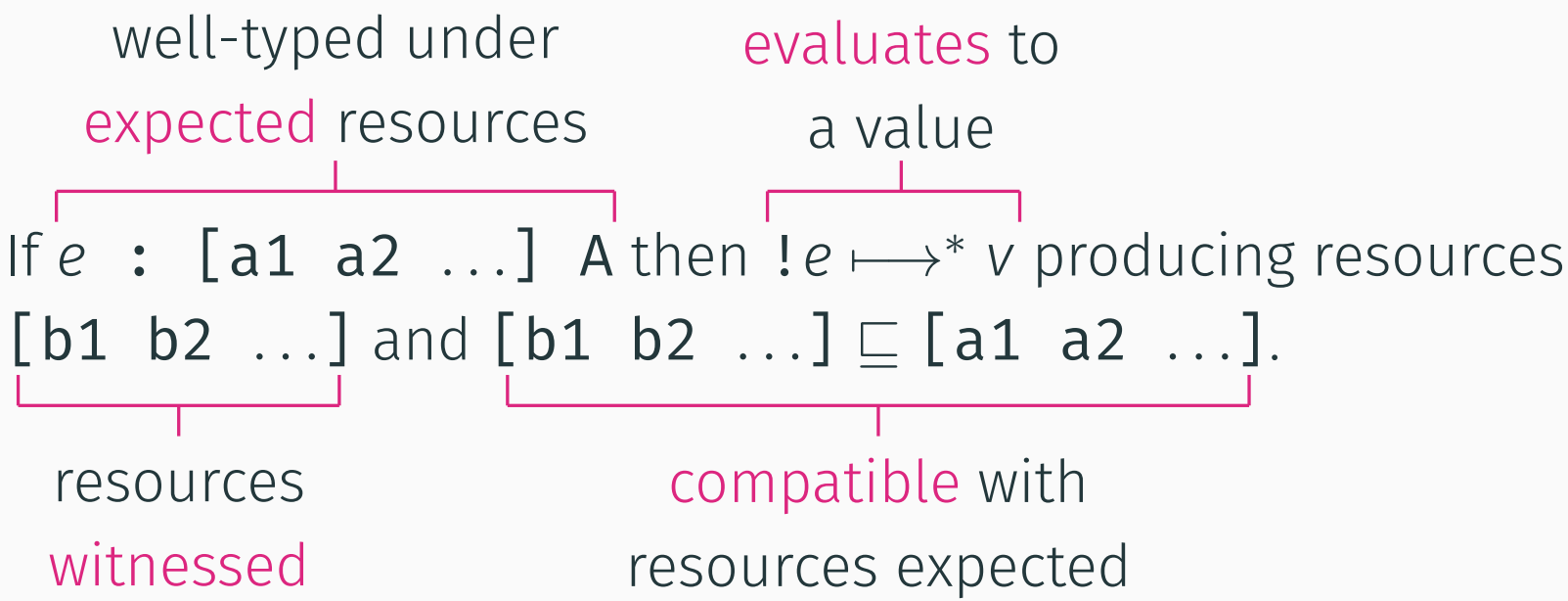
evaluates to a value

If e `: [a1 a2 ...] A` then `!`$e \longmapsto^* v$ producing resources `[b1 b2 ...]` and `[b1 b2 ...]` $\sqsubseteq$ `[a1 a2 ...]`.

Soundness Theorem



Soundness Theorem



Soundness Theorem $\Rightarrow$ Capability Safety

A token used as an identifier for an object such that ***possession of the token confers access rights for the object.*** *A capability can be thought of as a ticket.*

Soundness Theorem $\Rightarrow$ Capability Safety

A token used as an identifier for an object such that ***possession of the token confers access rights for the object.*** *A capability can be thought of as a ticket. Modification of a capability [...] is not allowable; however, unlike the case for tickets,* ***reproduction [...] is legal.***

Soundness Theorem $\Rightarrow$ Capability Safety

A token used as an identifier for an object such that ***possession of the token confers access rights for the object.*** *A capability can be thought of as a ticket. Modification of a capability [...] is not allowable; however, unlike the case for tickets,* ***reproduction [...] is legal.***

Soundness Theorem ⇒ Capability Safety

> *A token used as an identifier for an object such that **possession of the token confers access rights for the object.** A capability can be thought of as a ticket. Modification of a capability [...] is not allowable; however, unlike the case for tickets, **reproduction [...] is legal.***

If e `: [💎 a1 a2 ...] A` where 💎 is `strict` then `!`$e \longmapsto^* v$ produces resources `[b1 b2 ...]` $\ni$ 💎.

Soundness Theorem ⇒ Capability Safety

> *A token used as an identifier for an object such that **possession of the token confers access rights for the object.** A capability can be thought of as a ticket. Modification of a capability [...] is not allowable; however, unlike the case for tickets, **reproduction [...] is legal.***

If *e* `: [💎 a1 a2 ...] A` where 💎 is `strict` then `!`*e* $\longmapsto^*$ *v* produces resources `[b1 b2 ...]` $\ni$ 💎.

Soundness Theorem ⇒ Capability Safety

> *A token used as an identifier for an object such that **possession of the token confers access rights for the object.** A capability can be thought of as a ticket. Modification of a capability [...] is not allowable; however, unlike the case for tickets, **reproduction [...] is legal.***

If e `: [💎 a1 a2 ...] A` where 💎 is `strict` then `!`$e \longmapsto^* v$ produces resources `[b1 b2 ...]` $\ni$ 💎.

Soundness Theorem ⇒ Capability Safety

> *A token used as an identifier for an object such that **possession of the token confers access rights for the object.** A capability can be thought of as a ticket. Modification of a capability [...] is not allowable; however, unlike the case for tickets, **reproduction [...] is legal.***

If e `: [💎 a1 a2 ...] A` where 💎 is `strict` then $!e \longmapsto^* v$ produces resources `[b1 b2 ...]` $\ni$ 💎.

Soundness Theorem ⇒ Capability Safety

> *A token used as an identifier for an object such that **possession of the token confers access rights for the object.** A capability can be thought of as a ticket. Modification of a capability [...] is not allowable; however, unlike the case for tickets, **reproduction [...] is legal.***

If e `: [💎 a1 a2 ...] A` where 💎 is `strict` then `!`$e \longmapsto^* v$ produces resources `[b1 b2 ...]` $\ni$ 💎.

Soundness Theorem ⇒ Capability Safety

> *A token used as an identifier for an object such that **possession of the token confers access rights for the object.** A capability can be thought of as a ticket. Modification of a capability [...] is not allowable; however, unlike the case for tickets, **reproduction [...] is legal.***

If e `: [💎 a1 a2 ...] A` where 💎 is `strict` then `!`$e \longmapsto^* v$ produces resources `[b1 b2 ...]` $\ni$ 💎.

Proof Sketch

`[b1 b2 ...]` $\sqsubseteq$ `[💎 a1 a2 ...]` `[...]` $\not\sqsubseteq$ `[... 💎]`

Soundness Theorem ⇒ Capability Safety

> *A token used as an identifier for an object such that **possession of the token confers access rights for the object.** A capability can be thought of as a ticket. Modification of a capability [...] is not allowable; however, unlike the case for tickets, **reproduction [...] is legal.***

If e `: [💎 a1 a2 ...] A` where 💎 is `strict` then `!`$e \longmapsto^* v$ produces resources `[b1 b2 ...]` $\ni$ 💎.

Proof Sketch

`[b1 b2 ...]` $\sqsubseteq$ `[💎 a1 a2 ...]` $\qquad$ `[...]` $\not\sqsubseteq$ `[... 💎]`

by soundness

Soundness Theorem ⇒ Capability Safety

> *A token used as an identifier for an object such that **possession of the token confers access rights for the object.** A capability can be thought of as a ticket. Modification of a capability [...] is not allowable; however, unlike the case for tickets, **reproduction [...] is legal.***

If e `: [💎 a1 a2 ...] A` where 💎 is `strict` then `!`$e \longmapsto^* v$ produces resources `[b1 b2 ...]` $\ni$ 💎.

Proof Sketch

`[b1 b2 ...]` $\sqsubseteq$ `[💎 a1 a2 ...]` — by soundness

`[...]` $\not\sqsubseteq$ `[... 💎]` — by strictness

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔑 where 🔑 **`affine`** then
$!e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔑 where $k \leq n$.

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔐 where 🔐 **`affine`** then
$!e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔐 where $k \leq n$.

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔑 where 🔑 **`affine`** then
$!e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔑 where $k \leq n$.

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔐 where 🔐 `affine` then
`!`$e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔐 where $k \leq n$.

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔑 where 🔑 **`affine`** then
$!e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔑 where $k \leq n$.

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔐 where 🔐 **`affine`** then
$!e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔐 where $k \leq n$.

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔑 where 🔑 **`affine`** then
$!e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔑 where $k \leq n$.

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔑 where 🔑 **`affine`** then
$!e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔑 where $k \leq n$.

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔑 where 🔑 `affine` then
$!e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔑 where $k \leq n$.

Proof Sketch

$$\texttt{[b1 b2 ...]} \sqsubseteq \texttt{[a1 a2 ...]} \qquad [🔑\ 🔑] \not\sqsubseteq [🔑]$$

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔑 where 🔑 `affine` then
`!`$e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔑 where $k \leq n$.

Proof Sketch

`[b1 b2 ...]` $\sqsubseteq$ `[a1 a2 ...]` $\qquad$ [🔑 🔑] $\not\sqsubseteq$ [🔑]

by soundness

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔑 where 🔑 `affine` then
`!`$e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔑 where $k \leq n$.

Proof Sketch

`[b1 b2 ...]` $\sqsubseteq$ `[a1 a2 ...]` — by soundness

[🔑 🔑] $\not\sqsubseteq$ [🔑] — by affinity

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...] A` with n 🔐 where 🔐 `linear` then
$!e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔐 where $k = n$.

Proof Sketch

$$\underbrace{\texttt{[b1 b2 ...]} \sqsubseteq \texttt{[a1 a2 ...]}}_{\text{by soundness}} \qquad \underbrace{[🔐\ 🔐] \not\sqsubseteq [🔐]}_{\text{by affinity}}$$

Soundness Theorem ⇒ Quantity Safety

If e `: [a1 a2 ...]` A with n 🔑 where 🔑 `linear` then
`!`$e \longmapsto^* v$ produces `[b1 b2 ...]` with k 🔑 where $k = n$.

Proof Sketch

`[b1 b2 ...]` $\sqsubseteq$ `[a1 a2 ...]` — by soundness

`[`🔑 🔑`]` $\not\sqsubseteq$ `[`🔑`]` — by affinity

`[...]` $\not\sqsubseteq$ `[...` 🔑`]` — by strictness

Soundness Theorem $\Rightarrow$ Protocol Safety

If e `: [a1 a2 a3 ...] A` where `a1 a2` **`ordered`** then $!e \longmapsto^* v$ produces `[b1 b2 b3 ...]`.

Soundness Theorem $\Rightarrow$ Protocol Safety

If e `: [a1 a2 a3 ...] A` where `a1 a2 ordered` then $!e \longmapsto^* v$ produces `[b1 b2 b3 ...]`.

Soundness Theorem $\Rightarrow$ Protocol Safety

lacking all structural rules

If e `: [a1 a2 a3 ...] A` where `a1 a2 ordered` then $!e \longmapsto^* v$ produces `[a1 a2 b3 ...]`.

Soundness Theorem $\Rightarrow$ Protocol Safety

lacking all structural rules

If e `: [a1 a2 a3 ...] A` where `a1 a2 ordered` then $!e \longmapsto^* v$ produces `[a1 a2 b3 ...]`.

Proof Sketch: Analogous from soundness + weakening, contraction, exchange

More in the paper!

- ✌ Two further structural rules not mentioned here

More in the paper!

- ✌ Two further structural rules not mentioned here
- 🏗 Constructive Kripke semantics as a programming language

More in the paper!

- ✌️ Two further structural rules not mentioned here
- 🏗️ Constructive Kripke semantics as a programming language
- ⬇️⬆️ Shifting between substructual modes using quantification, correspondence to shifts in LNL + adjoint logic

More in the paper!

- ✌️ Two further structural rules not mentioned here
- 🏗️ Constructive Kripke semantics as a programming language
- ⬇️⬆️ Shifting between substructual modes using quantification, correspondence to shifts in LNL + adjoint logic
- 🧮 General proof technique capturing logical relations for open-ended effects

More in the paper!

- ✌️ Two further structural rules not mentioned here
- 🏗️ Constructive Kripke semantics as a programming language
- ⬇️⬆️ Shifting between substructual modes using quantification, correspondence to shifts in LNL + adjoint logic
- 🧮 General proof technique capturing logical relations for open-ended effects
- ‼️ More examples!

Takeaway: The effectful view on substructural reasoning newly unifies a set of old tools

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