The Holy Grail of Gradual Security

PL Wonks Seminar - Logic Seminar, Spring 2024

Tianyu Chen

Computer Science, Indiana University

° *The Achievement of the Grail.* Tapestries by Morris & Co, 1890. Birmingham Museum and Art Gallery

Road Map

- ☞ Background:
  - Information flow properties and noninterference
  - Information flow control: static, dynamic, gradual
  - The gradual guarantee
  - The tension between noninterference and the gradual guarantee
- $\lambda^{\star}_{\mathtt{IFC}}$ in Action
  - ✯ Solving the Tension Between Noninterference and the Gradual Guarantee
  - Type-Based Reasoning in $\lambda^{\star}_{\mathtt{IFC}}$
- Coercion-based Semantics for Gradual Security
- Meta-theoretical Results of $\lambda^{\star}_{\mathtt{IFC}}$

Information-Flow and Noninterference

Consider boolean negation. Can we infer `output` from `input`?

$$\texttt{let output} = \neg\texttt{input}$$

Yes! ✓

- Requires witnessing at least two executions:
  `input` = `true`, `output` = `false` and `input` = `false`, `output` = `true`
- An information-flow property is a hyperproperty: a predicate on sets of executions
- Noninterference: [1] two successful executions of a program produce the same value for different input.

[1] Specifically, termination-insensitive noninterference (TINI) for a functional programming language

Information-Flow Control (IFC)

- Security labels. Label input as `high`; output as `low`.

  Track and check the security labels
- IFC in a programming language, traditionally
  - Static: using a type system
  - Dynamic: using runtime monitoring
- A *gradual* security programming language
  - embeds both static and dynamic IFC
  - enables seamless transition between static and dynamic
  - security label annotations $\begin{cases} \text{specific: } \texttt{low}, \texttt{high} \\ \text{statically unknown: } \star \end{cases}$

Review: Static IFC Using a Type System

Modeling input and output:

$\texttt{user-input} : \texttt{Unit}_{\texttt{low}} \to \texttt{Bool}_{\texttt{high}}$ $\qquad$ $\texttt{publish} : \texttt{Bool}_{\texttt{low}} \to \texttt{Unit}_{\texttt{low}}$

Consider a statically-typed program:

```
let fconst = λ b : Bool_high. false in
let input  = user-input () in
let result = fconst input in
  publish result
```

✓ Well-typed *and* ✓ Runs successfully to `unit`

- Why? The return value of `fconst` is $\begin{cases} \text{always } \texttt{false} \\ \text{of } \texttt{low}\text{-security} \end{cases}$
- No runtime check!
- IFC enforced by the type system alone

Guarding Against Illegal Explicit Flows, Statically

Consider another fully static program:

```
let fid    = λ b : Bool_low . b in
let input  = user-input () in
let result = fid input in     // error
  publish result
```

✗ Ill-typed. Why?

- *Illegal* explicit flow from the `high`-security input to `fid`
  - `fid` expects `low` argument

✲ Program rejected by type-checker. Illegal explicit flow ruled out at compile time

Guarding Against Illegal Implicit Flows, Statically

Different observable behaviors in different branches:

```
let flip : Bool_high → Bool_low =
  λ b : Bool_high. if b then false else true in
let input  = user-input () in
let result = flip input in
  publish result
```

✗ Ill-typed

- Security label on the type of `if` is the join of its branches (both `low`) and the branch condition (`high`).
  - Expected: `low` (from type annotation $\texttt{Bool}_{\texttt{low}}$)
  - Actual: `high` (because of conditional)

✲ `high` $\not\leq$ `low`, thus rejected by type-checker. Illegal implicit flow ruled out at compile time

Guarding Against Illegal Explicit Flows, Dynamically

Consider the following dynamically-typed `fid` example that could potentially leak information through explicit flow:

```
let fid    = λ b : Bool⋆ . b in
let input  = user-input () in
let result = fid input in      // error
  publish result
```

✓ Well-typed *but* ✗ Fails at runtime

- The program errors regardless of input
- A runtime happens before the call to `publish` and checks whether `high` can flow to `low` (of course, no)

✲ Illegal explicit flow ruled out at runtime

[1] We annotate `Bool⋆` explicitly, which conforms with the syntax of $\lambda^{\star}_{\texttt{IFC}}$

Against Illegal Implicit Flows, Dynamically

Consider the following dynamically-typed `flip` example that could potentially leak information through implicit flow:

```
let flip : Bool⋆ → Bool⋆ =
  λ b : Bool⋆ . if b then false else true in
let input  = user-input () in
let result = flip input in
  publish result
```

✓ Well-typed *but* ✗ Fails at runtime

- The program (again) errors regardless of input
- `flip` produces a `high` value because of `high` branch condition
- A runtime happens before the call to `publish` and checks whether `high` can flow to `low` (of course, no)

✻ Illegal implicit flow ruled out at runtime

Gradual Typing Bridges Static and Dynamic IFC

Consider the following partially annotated version of `flip`. The return value must be `low`, because we intend to output the result:

```
let flip : Bool⋆ → Bool_low =
  λ b : Bool⋆ . if b then false else true in
let input  = user-input () in
let result = flip input in
  publish result
```

✓ Well-typed *but* ✗ Fails at runtime (for both `true` and `false`)

- ○ thus preventing the leak through implicit flow

- The information flow violation is detected *ealier* than the dynamic version, as `flip` returns

✻ Checking happens on the boundaries between statically- and dynamically-typed code fragments

The Gradual Guarantee

- Removing annotations from a correctly running program:

  Example: $(42 : \texttt{low} : \texttt{high}) \sqsupseteq (42 : \star : \texttt{high}) \sqsupseteq (42 : \star : \star)$

  → ... results in the same runtime behavior (42)

- Adding annotations may introduce more errors:

  Example: $(42 : \star : \star : \star) \sqsubseteq (42 : \texttt{high} : \star : \texttt{low})$

  ○ $(42 : \star : \star : \star) \Downarrow 42$ but $(42 : \texttt{high} : \star : \texttt{low}) \Downarrow \texttt{error}$

Satisfying Noninterference and the Gradual Guarantee in One Programming Language

... is hard according to the literature:

"We believe that there might be an inherent incompatibility between the strictness required to enforce a hyper-property like noninterference, and the optimistic flexibility dictated by the dynamic gradual guarantee."

Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References

"There is some recent evidence that the dynamic gradual guarantee – which some see as essential to gradual typing – is incompatible with various hyperproperties, like noninterference and parametricity."

Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing

Review: No-Sensitive-Upgrade Checking

- No-sensitive-upgrade (NSU) (Austin and Flanagan 2009) prevents implicit flow leaks through writes to mutable references
- For gradual typing, NSU happens at runtime, when type information is insufficient in deciding if a heap write is secure
- Program that potentially leaks information through the heap:

```
let input : Bool⋆ = user-input () in
let a       = ref low true in
  if input then a := false else a := true ;
  publish (! a)
```

✓ Well-typed *but* ✗ Fails at runtime (for both `true` and `false`)

- NSU checking terminates this program, because it attempts to write to a `low` memory location under a `high` execution context (PC), thus preventing the leak through heap

The Tension (in a Nutshell)

Toro et al. [2018] discover a tension between noninterference and the gradual guarantee in their language design, $\mathsf{GSL}_{\mathsf{Ref}}$.

Counterexample of the gradual guarantee in $\mathsf{GSL}_{\mathsf{Ref}}$:

Left: less precise, more dynamic

```
let x = user-input () in
let y = ref Bool⋆ true⋆ in
  if x then (y := false_high)
       else ()
```

Right: more precise, more static

```
let x = user-input () in
let y = ref Bool_high true_high in
  if x then (y := false_high)
       else ()
```

- ✓ Both are well-typed
- ✓ The more precise (Right) program runs successfully to `unit`
- ✗ The less precise (Left) program errors!
  - In $\mathsf{GSL}_{\mathsf{Ref}}$, ⋆ corresponds to the interval [`low`, `high`]
- ✖ Violates the gradual guarantee!

Possible Sources of the Tension

Lang.	Noninter-ference	Gradual Guarantee	Type-guided classification	NSU	Runtime security labels
GSL_{Ref}	✓	✗	✓	✓	{low, high, ⋆}
GLIO	✓	✓	✗	✓	{low, high}
WHILE^{G}	✓	✓	✓	✗	{low, high, ⋆}
$\lambda^{\star}_{\texttt{IFC}}$ (ours)	✓	✓	✓	✓	{low, high}

Road Map

- Background
- ☞ $\lambda^{\star}_{\mathtt{IFC}}$ in Action:
  - ✯ Solving the Tension Between Noninterference and the Gradual Guarantee
  - ○ Type-Based Reasoning in $\lambda^{\star}_{\mathtt{IFC}}$
- Coercion-based Semantics for Gradual Security
- Meta-theoretical Results of $\lambda^{\star}_{\mathtt{IFC}}$

Solution to the Tension, in $\lambda^{\star}_{\mathtt{IFC}}$

Left: less precise, more dynamic

```
let x = user-input () in
let y : (Ref Bool_⋆)_⋆ =
    ref high true_high in
  if x then (y := false_high)
        else ()
```

Right: more precise, more static

```
let x = user-input () in
let y : (Ref Bool_high)_high =
    ref high true_high in
if x then (y := false_high)
     else ()
```

- ✓ Both are well-typed
- ✓ The more precise (Right) program runs successfully to `unit`
- ✓ The less precise (Left) one also runs successfully to `unit`
- ✻ Does *not* violate the gradual guarantee!

Problem solved!

But why?

Less precise in GSL_{Ref}:

```
let x = user-input () in
let y = ref Bool_★ true_★ in
  if x then (y := false_high)
       else ()
```

Less precision in $\lambda^{\star}_{\texttt{IFC}}$:

```
let x = user-input () in
let y : (Ref Bool_★)_★ =
    ref high true_high in
  if x then (y := false_high)
       else ()
```

More precise in GSL_{Ref}:

```
let x = user-input () in
let y = ref Bool_high true_high in
  if x then (y := false_high)
       else ()
```

More precise in $\lambda^{\star}_{\texttt{IFC}}$:

```
let x = user-input () in
let y : (Ref Bool_high)_high =
    ref high true_high in
  if x then (y := false_high)
       else ()
```

In $\lambda^{\star}_{\texttt{IFC}}$, Security labels on type annotations can be specific or ★, but those on literals and memory locations stay specific.

Omitted security label annotations on literals default to `low`:

Less precise in GSL_{Ref}:

```
let x = user-input () in
let y = ref Bool_⋆ true_⋆ in
  if x then (y := false_high)
       else ()
```

Less precision in $\lambda^{\star}_{\text{IFC}}$:

```
let x = user-input () in
let y : (Ref Bool_⋆)_⋆ =
    ref high true in
  if x then (y := false)
       else ()
```

Solving the Tension in $\lambda^{\star}_{\mathtt{IFC}}$ (Summary)

Design choices of $\mathrm{GSL}_{\mathsf{Ref}}$:

- Security labels on both types and literals can be $\star$
- Runtime security labels can also be $\star$ (due to $\star$ on literals)
- Runtime has to "guess" conservatively
  - → more runtime errors when moving toward less precise
  - → violates the gradual guarantee!

Design choices of $\lambda^{\star}_{\mathtt{IFC}}$:

- Security labels on type annotations may decrease in precision $(\mathtt{Ref\ Bool}_{\star})_{\star} \sqsubseteq (\mathtt{Ref\ Bool}_{\mathtt{high}})_{\mathtt{high}}$
  - NSU checking happens. Heap IFC policy enforced at runtime
- Labels on literals and memory locations remain specific
  - security of data: only the programmer knows; must not be inferred
  - → runtime security levels remain specific during program execution

Security Coercions as Runtime IFC Monitor

Revisit the dynamically-typed $\lambda^{\star}_{\texttt{IFC}}$ program:

```
let flip : Bool⋆ → Bool⋆ =
    λ b : Bool⋆ . if b then false else true in
let input  = user-input () in
let result = flip input in
  publish result
```

Compile the $\lambda^{\star}_{\texttt{IFC}}$ program to the following cast calculus $\lambda^{c}_{\texttt{IFC}}$ term, by making all casts explicit:

```
let flip = λ b . if b then (false ⟨low!⟩)
                          else (true ⟨low!⟩) in
let input  = user-input () in
let result = flip (input ⟨high!⟩) in
  publish (result ⟨low ?^p⟩)
```

Reducing the $\lambda^{c}_{\mathtt{IFC}}$ term blames the projection (before calling publish):

$$
\begin{array}{l}
\texttt{let result = ((}\lambda\texttt{ b. if b then (false}\langle\texttt{low}\,!\rangle\texttt{) else ...)} \\
\qquad\qquad\texttt{(true}\langle\texttt{high}\,!\rangle\texttt{)) in} \\
\quad\texttt{publish (result}\langle\texttt{low}\,?^{p}\rangle\texttt{)}
\end{array} \longrightarrow^{*} \qquad (1)
$$

$$
\begin{array}{l}
\texttt{let result = prot low (if (true}\langle\texttt{high}\,!\rangle\texttt{)} \\
\qquad\qquad\qquad\qquad\texttt{then (false}\langle\texttt{low}\,!\rangle\texttt{) else ...) in} \\
\quad\texttt{publish (result}\langle\texttt{low}\,?^{p}\rangle\texttt{)}
\end{array} \longrightarrow^{*} \qquad (2)
$$

$$
\begin{array}{l}
\texttt{let result = prot low (prot high (false}\langle\texttt{low}\,!\rangle\texttt{)) in} \\
\quad\texttt{publish (result}\langle\texttt{low}\,?^{p}\rangle\texttt{)}
\end{array} \longrightarrow^{*} \qquad (3)
$$

$$
\begin{array}{l}
\texttt{let result = prot low (false}\langle\uparrow\texttt{;}\,\texttt{high}\,!\rangle\texttt{) in} \\
\quad\texttt{publish (result}\langle\texttt{low}\,?^{p}\rangle\texttt{)}
\end{array} \longrightarrow^{*} \qquad (4)
$$

$$
\longrightarrow^{*} \texttt{publish (false}\langle\uparrow\texttt{;}\,\texttt{high}\,!\texttt{;}\,\texttt{low}\,?^{p}\rangle\texttt{)} \qquad (5)
$$

$$
\longrightarrow^{*} \texttt{blame}\ p \qquad (6)
$$

Sequencing models explicit flow. Stamping models implicit flow.
Checking by reducing coercion sequences

Type-Based Reasoning in $\lambda^{\star}_{\mathtt{IFC}}$

- Type-based reasoning: Toro et al. [2018] observe that security typing induces "free theorems" about noninterference
- Type-based reasoning is the synergy of two design choices:
  1. Vigilance
  2. Type-Guided Classification
- GLIO (Azevedo de Amorim et al. 2020) satisfies the gradual guarantee by sacrificing type-guide classification, which they claim to be the reason $\mathrm{GSL}_{\mathsf{Ref}}$(Toro et al. 2018) violates the gradual guarantee
- $\lambda^{\star}_{\mathtt{IFC}}$ supports type-based reasoning just like $\mathrm{GSL}_{\mathsf{Ref}}$

Vigilance: Type-Based Reasoning for Explicit Flows

Consider the example from Toro et al. [2018]:

```
let mix : Int_low → Int_high → Int_low =
    λ pub priv .
      if pub < (priv : Int⋆ : Int_low) then 1 else 2 in
  mix 1_low 5_low
```

Free theorem: Either ① the low result of mix never depends on the high priv argument or ② mix produces a runtime error.

(GLIO: not vigilant → does not produce an error → violates the free theorem)

In $\lambda^{\star}_{\mathtt{IFC}}$, $\mathtt{5}\,\langle \uparrow ;\, \mathtt{high}\,!\,;\, \mathtt{low}\,?^{p} \rangle \Downarrow \mathtt{blame}\ p$

Type-Guided Classification: Type-Based Reasoning for Implicit Flows

Another example from Toro et al. [2018]:

```
let mix : Int_low → Int_⋆ → Int_low =
    λ pub priv. if pub < priv then 1 else 2 in
let smix : Int_low → Int_high → Int_low =
    λ pub priv. mix pub priv in
  smix 1_low 5_low
```

Free theorem: The `smix` function either ① returns a value that does not depend on `priv` or ② produces a runtime error

(GLIO: ① not vigilant ② does not classify values using types → does not produce an error → violates the free theorem)

```
let mix = λ pub priv.
  (if (pub ⟨low!⟩) < priv
      then (1 ⟨low!⟩)
      else (2 ⟨low!⟩)) ⟨low?^p⟩ in
let smix = λ pub priv. mix pub (priv ⟨high!⟩) in
  smix 1 (5 ⟨↑⟩)
```

$$\longrightarrow^* \quad \texttt{(if (1}\langle\texttt{low!}\rangle \texttt{ < 5}\langle\uparrow;\texttt{high!}\rangle\texttt{) then 1}\langle\texttt{low!}\rangle\texttt{ else ...)}\langle\texttt{low?}^{p}\rangle \tag{7}$$

$$\longrightarrow^* \quad \texttt{(if (true}\langle\uparrow;\texttt{high!}\rangle\texttt{) then 1}\langle\texttt{low!}\rangle\texttt{ else ...)}\langle\texttt{low?}^{p}\rangle \tag{8}$$

$$\longrightarrow^* \quad \texttt{(prot high (1}\langle\texttt{low!}\rangle\texttt{))}\langle\texttt{low?}^{p}\rangle \tag{9}$$

$$\longrightarrow^* \quad \texttt{1}\langle\uparrow;\texttt{high!}\rangle\langle\texttt{low?}^{p}\rangle \tag{10}$$

$$\longrightarrow^* \quad \texttt{blame } p \tag{11}$$

In $\lambda^{\star}_{\texttt{IFC}}$, the program errors, thus satisfying the free theorem

Road Map

- Background
- $\lambda^{\star}_{\texttt{IFC}}$ in Action
- ☞ Coercion-based Semantics for Gradual Security
- Meta-theoretical Results of $\lambda^{\star}_{\texttt{IFC}}$

Coercion Calculus for Security Labels

Syntax and typing for security coercions and coercion sequences:

specific security labels	ℓ	$\in$	$\{\texttt{low}, \texttt{high}\}$
gradual security labels	g	$::=$	$\star \mid \ell$
blame labels	p, q		
security coercions	c, d	$::=$	$\mathrm{id}(g) \mid \uparrow \mid \ell\,! \mid \ell\,?^p \mid \bot^p$
coercion sequences	$\bar{c}, \bar{d}$	$::=$	$\mathrm{id}(g) \mid \bot^p\, g_1\, g_2 \mid \bar{c}\,;\, c$

$\boxed{\vdash c : g_1 \Rightarrow g_2}$

$$\frac{}{\vdash \mathrm{id}(g) : g \Rightarrow g} \qquad \frac{}{\vdash \uparrow : \texttt{low} \Rightarrow \texttt{high}} \qquad \frac{}{\vdash \ell\,! : \ell \Rightarrow \star}$$

$$\frac{}{\vdash \ell\,?^p : \star \Rightarrow \ell} \qquad \frac{}{\vdash \bot^p : \texttt{high} \Rightarrow \texttt{low}}$$

Reduction semantics and normal forms of the coercion calculus on security labels:

$\boxed{\mathrm{NF}\ \bar{c}}$

$$\frac{}{\mathrm{NF}\ \mathrm{id}(g)} \qquad \frac{}{\mathrm{NF}\ \mathrm{id}(\star)\ ;\ \ell\ ?^{p}} \qquad \frac{\mathrm{NF}\ \bar{c}}{\mathrm{NF}\ \bar{c}\ ;\ \ell\ !} \qquad \frac{\mathrm{NF}\ \bar{c}}{\mathrm{NF}\ \bar{c}\ ;\ \uparrow}$$

$\boxed{c\ ;\ c \longrightarrow c}$

$$?\text{-}id\ \frac{}{\ell\ !\ ;\ \ell\ ?^{p} \longrightarrow \mathrm{id}(\ell)} \qquad ?\text{-}\uparrow\ \frac{}{\texttt{low}\ !\ ;\ \texttt{high}\ ?^{p} \longrightarrow \uparrow}$$

$$?\text{-}\bot\ \frac{}{\texttt{high}\ !\ ;\ \texttt{low}\ ?^{p} \longrightarrow \bot^{p}}$$

$\boxed{\bar{c} \longrightarrow \bar{d}}$

$$id\ \frac{\mathrm{NF}\ \bar{c}}{\bar{c}\ ;\ \mathrm{id}(g) \longrightarrow \bar{c}} \qquad \bot\ \frac{\mathrm{NF}\ \bar{c} \qquad \vdash \bar{c} : g_1 \Rightarrow g_2}{\bar{c}\ ;\ \bot^{p} \longrightarrow \bot^{p}\ g_1\ \texttt{low}}$$

$$\xi\text{-}\bot\ \frac{\vdash c : g_2 \Rightarrow g_3}{\bot^{p}\ g_1\ g_2\ ;\ c \longrightarrow \bot^{p}\ g_1\ g_3}$$

$$\xi_L\ \frac{\bar{c} \longrightarrow \bar{d}}{\bar{c}\ ;\ c \longrightarrow \bar{d}\ ;\ c} \qquad \xi_R\ \frac{\mathrm{NF}\ \bar{c} \qquad c\ ;\ d \longrightarrow c'}{\bar{c}\ ;\ c\ ;\ d \longrightarrow \bar{c}; c'}$$

(A Glimpse of) the Cast Calculus $\lambda^{c}_{\mathtt{IFC}}$

- Representation of PC: label expressions

  $e, PC ::= \ell \mid \texttt{blame}\ p \mid e\langle \bar{c} \rangle$

- Coercions on values of $\lambda^{c}_{\mathtt{IFC}}$:

base types	ι	$::=$	$\texttt{Unit} \mid \texttt{Bool}$
raw types	T, S	$::=$	$\iota \mid A \xrightarrow{gc} B \mid \texttt{Ref}\ (T_g)$
types	A, B	$::=$	T_g
raw coercions	c_r, d_r	$::=$	$\mathrm{id}(\iota) \mid \mathrm{Ref}\ \boldsymbol{c}\ \boldsymbol{d} \mid (\bar{d}, \boldsymbol{c} \rightarrow \boldsymbol{d})$
coercions	$\boldsymbol{c}, \boldsymbol{d}$	$::=$	$c_r, \bar{c}$

- NSU checking: reducing label expressions

$$\frac{n\ \mathsf{FreshIn}\ \mu(\ell) \qquad PC\langle \star \Rightarrow^{p} \ell \rangle \longrightarrow^{*} PC'}{\texttt{ref?}^{p}\ \ell\ V \mid \mu \mid PC \longrightarrow \texttt{addr}\ n \mid (\mu, \ell \mapsto n \mapsto V)}$$

$$\frac{\mathsf{NF}\ \bar{c} \qquad (\mathit{stamp}!\ PC\ |\bar{c}|)\langle \star \Rightarrow^{p} \hat{\ell} \rangle \longrightarrow^{*} PC' \qquad V\langle \boldsymbol{c} \rangle \longrightarrow^{*} W}{\texttt{assign?}^{p}\ (\texttt{addr}\ n\langle \mathrm{Ref}\ \boldsymbol{c}\ \boldsymbol{d}, \bar{c} \rangle)\ V\ T\ g \mid \mu \mid PC \longrightarrow \$\ \texttt{unit} \mid [\hat{\ell} \mapsto n \mapsto W]\ \mu}$$

$$\vdash \boldsymbol{c} : T_g \Rightarrow S_{\hat{\ell}}, \vdash \boldsymbol{d} : S_{\hat{\ell}} \Rightarrow T_g$$

Road Map

- Background
- $\lambda^{\star}_{\mathtt{IFC}}$ in Action
- Coercion-based Semantics for Gradual Security
- ☞ Meta-theoretical Results of $\lambda^{\star}_{\mathtt{IFC}}$

Theorem (Compilation preserves types)

If $\Gamma; g \vdash M : A$, *then* $\Gamma; \varnothing; g; \texttt{low} \vdash \mathcal{C}\ M : A$.

Theorem (Progress)

Suppose PC is well-typed: $\vdash PC \Leftarrow g$, M *is well-typed:* $\varnothing; \Sigma; g; |PC| \vdash M \Leftarrow A$, *and the heap* μ *is well-typed:* $\Sigma \vdash \mu$. *Then either (1)* M *is a value or (2)* M *is a blame or (3)* M *can take a reduction step:* $M \mid \mu \mid PC \longrightarrow N \mid \mu'$ *for some* N *and* μ'.

Theorem (Preservation)

Suppose PC is well-typed: $\vdash PC \Leftarrow g$, M *is well-typed:* $\varnothing; \Sigma; g; |PC| \vdash M \Leftarrow A$ *and the heap* μ *is well-typed:* $\Sigma \vdash \mu$. *If* $M \mid \mu \mid PC \longrightarrow N \mid \mu'$, *there exists* Σ' *s.t* $\Sigma' \supseteq \Sigma$, $\varnothing; \Sigma'; g; |PC| \vdash N \Leftarrow A$, *and* $\Sigma' \vdash \mu'$.

Theorem (The gradual guarantee)

Suppose M, M' are related by precision:

$$\varnothing; \varnothing; \varnothing; \varnothing; \texttt{low}; \texttt{low}; \texttt{low}; \texttt{low} \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A'$$

If M' evaluations to a value:

$$M' \mid \varnothing \mid \texttt{low} \longrightarrow^* V' \mid \mu'$$

there exists V and μ s.t. M evaluates to V:

$$M \mid \varnothing \mid \texttt{low} \longrightarrow^* V \mid \mu$$

and the resulting values are related by precision for some Σ, Σ':

$$\varnothing; \varnothing; \Sigma; \Sigma'; \texttt{low}; \texttt{low}; \texttt{low}; \texttt{low} \vdash V \sqsubseteq V' \Leftarrow A \sqsubseteq A'$$

The noninterference of $\lambda^\star_{\texttt{IFC}}$ is conjectured by that of $\lambda^\star_{\texttt{SEC}}$:

$\lambda^\star_{\texttt{IFC}}$ performs type-guided classification but $\lambda^\star_{\texttt{SEC}}$ does not, so the value that a $\lambda^\star_{\texttt{IFC}}$ program produces is at least as secure as the value produced by the same program in $\lambda^\star_{\texttt{SEC}}$.

Theorem (Noninterference of $\lambda^\star_{\texttt{SEC}}$)

If M is well-typed $(x{:}\texttt{Bool}_{\texttt{high}}); \varnothing; \texttt{low}; \texttt{low} \vdash M : \texttt{Bool}_{\texttt{low}}$ *and*

$$\varnothing \mid \texttt{low} \vdash M[x := (b_1)_{\texttt{high}}] \Downarrow V_1 \mid \mu_1$$
$$\varnothing \mid \texttt{low} \vdash M[x := (b_2)_{\texttt{high}}] \Downarrow V_2 \mid \mu_2$$

then $V_1 = V_2$.

Code and Data Availability

https://github.com/Gradual-Typing/LambdaSecStar

```
tianyu@beluga:~/workspace/agda/LambdaSecStar

> tianyu @ beluga in ~/workspace/agda/LambdaSecStar on git:master v [17:15:00]
$ polyglot --exclude plot/ --exclude notes/ --exclude bin/ --exclude _build/ --exclude src/CCExpSub .
-------------------------------------------------------------------------------
 Language            Files        Lines         Code     Comments       Blanks
-------------------------------------------------------------------------------
 Agda                  201        22618        19774          288         2556
 Makefile                1           20           15            0            5
 Markdown                4          182          148            0           34
-------------------------------------------------------------------------------
 Total                 206        22820        19937          288         2595
-------------------------------------------------------------------------------

> tianyu @ beluga in ~/workspace/agda/LambdaSecStar on git:master v [17:15:06]
$ 
```

Main Takeaways

1. It is possible to satisfy both noninterference and the gradual guarantee in a gradual security-typed language, provided that the security level of data remains specific at runtime
2. Gradual information flow can be represented as coercions. In particular, NSU checking is a special projection that casts PC to the security of the memory location to modify
3. The key to the semantics design of of a gradual security-typed language is identifying injections ($\ell\,!$) and projections ($\ell\,?^{p}$)

Thank you for your attention!