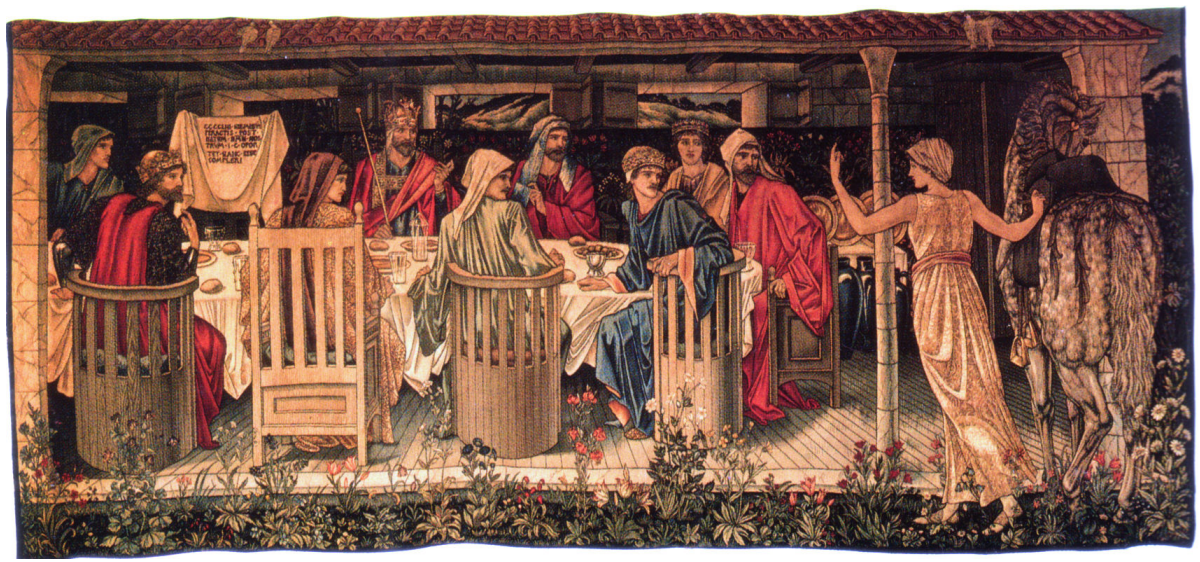
The Holy Grail of Gradual Security

Final Examination for Doctor of Philosophy in Computer Science

Tianyu Chen

Indiana University



° *The Summons*.The Holy Grail tapestries by Morris & Co. Birmingham Museum and Art Gallery

Tianyu's Thesis Statement

It *is possible* to design a gradual IFC programming language that satisfies both noninterference and the gradual guarantee while supporting type-based reasoning, by excluding the unknown label $\star$ from run-time security labels and using security coercions to represent casts.

Road Map

- ☞ Background
  - Explicit flow and implicit flow
  - Information flow control (IFC): static, dynamic, and gradual
  - The gradual guarantee and its tension with IFC
  - Source of the tension: including $\star$ in runtime labels
- $\lambda^{\star}_{\mathtt{IFC}}$: a gradual IFC calculus
  - $\lambda^{\star}_{\mathtt{IFC}}$ enforces IFC while satisfying gradual guarantee
  - $\lambda^{\star}_{\mathtt{IFC}}$ supports type-based reasoning (free theorems)
- Technical development
  - Formal definition of $\lambda^{\star}_{\mathtt{IFC}}$
  - Coercion calculi for IFC
  - The IFC cast calculus $\lambda^{c}_{\mathtt{IFC}}$
- Meta-theoretic results
  - Type safety for $\lambda^{\star}_{\mathtt{IFC}}$
  - Gradual guarantee for $\lambda^{\star}_{\mathtt{IFC}}$
  - Noninterference for $\lambda^{\star}_{\mathtt{IFC}}$

Explicit Information Flow

Can we infer input from output in the following program?

```
let input = private-input () in
  publish (¬ input)
```

Explicit Information Flow

Can we infer input from output in the following program?

```
let input = private-input () in
  publish (¬ input)
```

✓ Yes!

- Witness at least two executions
- Output is the negation of input
- Explicit flow

Implicit Information Flow

Can we infer input from output in the following program?

```
let input = private-input () in
    publish (if input then false else true)
```

Implicit Information Flow

Can we infer input from output in the following program?

```
let input = private-input () in
    publish (if input then false else true)
```

✓ Also yes

- Again, output is the negation of input
- Implicit flow: input influences output through *branching*

Information-Flow Control (IFC)

- Ensures that information transfers adhere to a security policy
- For example, `high` input must not flow to `low` output
- Propagate and check the security labels
- IFC in PL { **static** using a type system; **dynamic** using runtime monitoring

Static IFC Accepts Legal Explicit Flow

(Static IFC using a type system)

```
1 let fconst = λ b : Bool_high. false in
2 let input  = private-input () in
3 let result = fconst input in
4   publish result
```

✓ Well-typed and runs successfully to `unit`

- Why? The return value of `fconst` is { always `false` / of `low`-security
- Accepted by type-checker. No runtime check

°`private-input : Unit_low → Bool_high` and `publish : Bool_low → Unit_low`

Static IFC Rejects Illegal Explicit Flow

(Replace `fconst` with `flip`)

```
1 let flip   = λ b : Bool_low . ¬ b in
2 let input  = private-input () in
3 let result = flip input in  // compilation error
4   publish result
```

✗ Ill-typed. Illegal explicit flow:

- input is `high`
- `flip` expects `low` argument

▸ Rejected by type-checker. Again no runtime check

Dynamic Enforcement of Explicit Flow

(Revisit `flip` with dynamic IFC)

```
1 let flip    = λ b. ¬ b in
2 let input   = private-input () in
3 let result  = flip input in
4   publish result    // runtime error
```

✗ Errors at runtime (regardless of input)

- A runtime check happens before calling `publish`

In dynamic IFC, runtime values are tagged with their security level. The labels can originate from

- primitive operations
- annotations on literals
- the security level of the execution context

Static Enforcement of Implicit Flow

(Different behavior in different branches)

```
1 let flip : Bool_high → Bool_low =
2     λ b : Bool_high. if b then false else true in
3 let input  = private-input () in
4 let result = flip input in
5   publish result
```

✗ Ill-typed

- Security label on the type of `if` is the join (least upper bound) of its branches (`low`) and the branch condition (`high`).
- Rejected by type-checker. No runtime check

Dynamic Enforcement of Implicit Flow

(Enforcing implicit flow with dynamic IFC)

```
1 let flip   = λ b. if b then false else true in
2 let input  = private-input () in
3 let result = flip input in
4   publish result
```

✗ Errors at runtime (regardless of input)

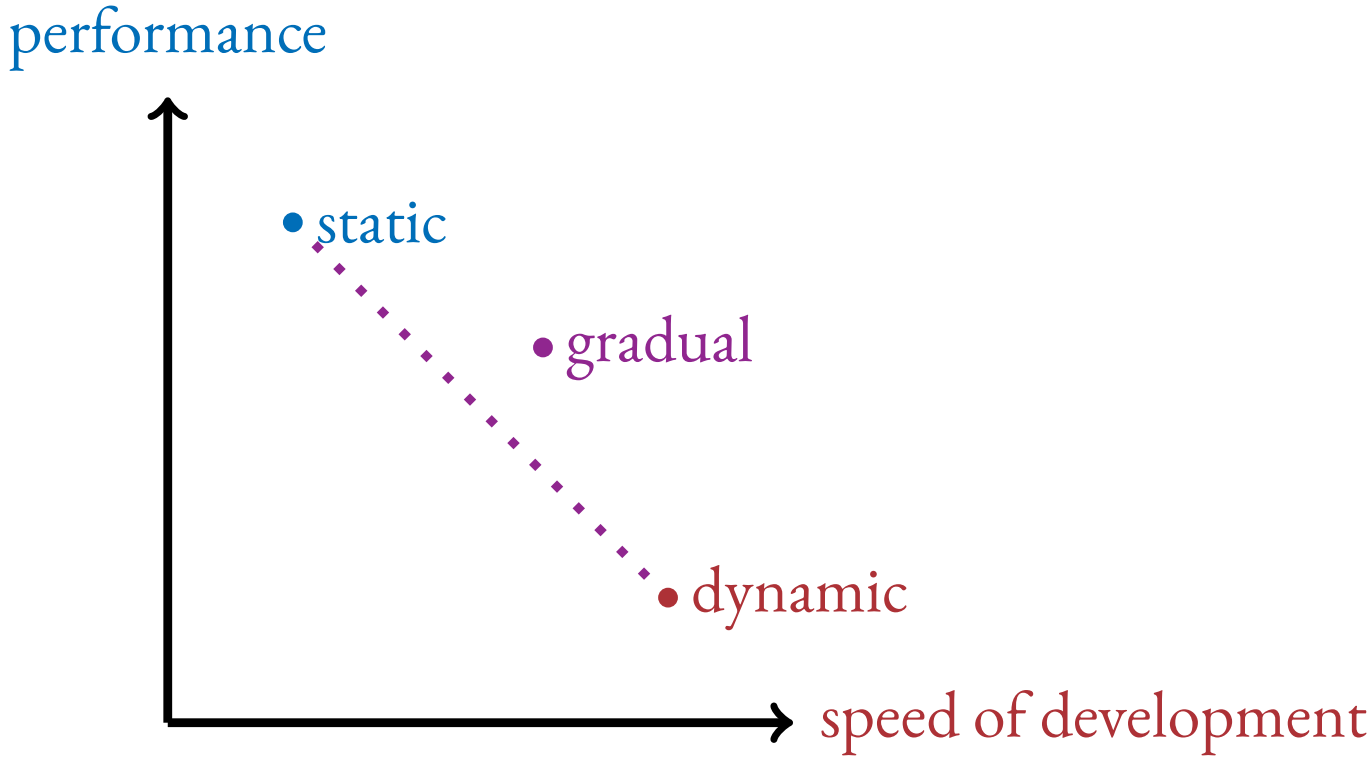
- `flip` produces a `high` value because of `high` branch condition
- A runtime check happens before calling `publish`
- Illegal implicit flow ruled out at runtime

Static IFC Is Hard to Use

"Another myth spread by security researchers is that the planet Earth contains more than six programmers who can correctly use security labels and information flow control (IFC)." [1]

[1] James Mickens. *This World of Ours.* Usenix ;login: 2014

Gradual IFC Comes to the Rescue!



In gradual IFC, static type information is optional:

- Go static in parts of the application where performance matters
- Go dynamic when performance matters less, and ease-of-use matters more

Principle of Locality: 90% of execution time in 10% of the code [2]

[2] Dr. Ranjani Parthasarathi. *Computer Architecture: Engineering And Technology*

Gradual Typing Bridges Static and Dynamic IFC

Partially-annotated `flip`:

```
1 let flip : Bool⋆ → Bool_low =
2     λ b : Bool⋆ . if b then false else true in
3 let input  = private-input () in
4 let result = flip input in
5   publish result
```

- Well-typed but errors at runtime
- Checking happens on the boundaries between static and dynamic fragments
- The information flow violation is detected earlier than the dynamic version, as `flip` returns.

[2] Blue: types with statically-known security. Red: types with unknown security.

The Gradual Guarantee

less precise

```
let f : Bool⋆ → Bool⋆ =
    λ b : Bool⋆ . true in
let i = private-input () in
let result = f i in
  publish result
```

⊑

```
let f : Bool⋆ → Bool_low =
    λ b : Bool⋆ . true in
let i = private-input () in
let result = f i in
  publish result
```

⊑

more precise

```
let f : Bool_high → Bool_low =
    λ b : Bool_high . true in
let i = private-input () in
let result = f i in
  publish result
```

- In the absense of errors, adding or removing security annotations does not change the result of the program.
- Adding security annotations may trigger errors.
- Removing security annotations may not trigger errors.

The Gradual Guarantee

less precise

```
let f : Bool⋆ → Bool⋆ =
    λ b : Bool⋆ . true in
let i = private-input () in
let result = f i in
  publish result
```

⊑

```
let f : Bool⋆ → Bool_low =
    λ b : Bool⋆ . true in
let i = private-input () in
let result = f i in
  publish result
```

⊑

more precise

```
let f : Bool_high → Bool_low =
    λ b : Bool_high . true in
let i = private-input () in
let result = f i in
  publish result
```

- In the absense of errors, adding or removing security annotations does not change the result of the program.
- Adding security annotations may trigger errors.
- Removing security annotations may not trigger errors.

There is a tension between the gradual guarantee and IFC!

☹

Static Enforcement of Flows Through Mutable References

```
1 let a     = ref low true in
2 let input = private_input () in
3 if input then
4     a := false
5 else
6     a := true
7 publish (! a)
```

- The reference has type $\texttt{Ref}\ (\texttt{Bool}_{\texttt{low}})$. It points to a low memory location.
- The type of the branch condition is $\texttt{Bool}_{\texttt{high}}$
- ✗ ~~Writing to low memory under a high branch condition~~

Dynamic Enforcement of Flows Through Mutable References

```
1 let a     = ref low true in
2 let input = private_input () in
3 if input then
4     a := false
5 else
6     a := true
7 publish (! a)
```

The assignments fail at runtime because the no-sensitive-upgrade (NSU) mechanism [3] prevents writing to a `low` security pointer in a `high` security branch.

[3] Austin and Flanagan. *Efficient purely-dynamic information flow analysis.* PLAS 2009.

Counterexample of Gradual Guarantee in GSL_{Ref}

less precise

```
1 let x = private-input () in
2 let a = ref ⋆ true⋆ in
3 if x then (a := false_high)
4      else ()
```

more precise

```
let x = private-input () in
let a = ref high true_high in
if x then (a := false_high)
     else ()
```

✓ The more precise program (right) runs successfully

✗ But the less precise version (left) errors in GSL_{Ref}[4]

- The assignment fails because it is in a high-security branch and GSL_{Ref} conservatively treats the reference's label ($\star$) as if it were `low`

[4] Toro, Garcia, Tanter. *Type-Driven Gradual Security with References.* TOPLAS 2018.

But wait... GSL_{Ref} allows $\star$ labels on values?

The counterexample depends on labeling a reference with unknown security ($\star$):

```
1 let x = private-input () in
2 let a = ref ⋆ true⋆ in
3 if x then (a := false_high)
4      else ()
```

- Dynamic IFC languages don't use $\star$ as a runtime security label.
- Gradual languages traditionally use $\star$ for type checking, but not for categorizing runtime values.
- The inputs to an information flow system are the user's choices regarding what data is high or low security.

Sources of the Tension with the Gradual Guarantee

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

Removing ⋆ from the runtime labels enables the gradual guarantee.

Road Map

- Background
- ☞ $\lambda^{\star}_{\mathtt{IFC}}$: a gradual IFC calculus
  - $\lambda^{\star}_{\mathtt{IFC}}$ enforces IFC while satisfying gradual guarantee
  - $\lambda^{\star}_{\mathtt{IFC}}$ supports type-based reasoning (free theorems)
- Technical development
- Meta-theoretic results

$\lambda^{\star}_{\mathtt{IFC}}$ Excludes $\star$ From Runtime Labels

less precise

```
1 let x = private-input () in
2 let a : (Ref Bool⋆)⋆ =
3     ref high true_high in
4 if x then (a := false_high)
5      else ()
```

more precise

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

✓ The more precise program runs successfully to `unit`

✓ The less precise program also runs successfully to `unit`

$\lambda^{\star}_{\mathsf{IFC}}$ Excludes $\star$ From Runtime Labels

less precise

```
1 let x = private-input () in
2 let a : (Ref Bool⋆)⋆ =
3     ref high true_high in
4 if x then (a := false_high)
5      else ()
```

more precise

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

- ✓ The more precise program runs successfully to `unit`
- ✓ The less precise program also runs successfully to `unit`
- ✓ **Problem solved!**

Comparing $\lambda^{\star}_{\mathtt{IFC}}$ With $\mathrm{GSL}_{\mathsf{Ref}}$

- The default security label in $\lambda^{\star}_{\mathtt{IFC}}$ is `low`, so the programmer does not have to label constants
- Remove the labels on constants to obtain the following program, which also reduces successfully to unit:

```
let x = private-input () in
let a : (Ref Bool⋆)⋆ = ref high true in
if x then (a := false)
     else ()
```

Comparing with the program in $\mathrm{GSL}_{\mathsf{Ref}}$, which errors:

```
let x = private-input () in
let a = ref ⋆ true⋆ in
if x then (a := falsehigh)
     else ()
```

Vigilance: Type-Based Reasoning for Explicit Flows

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

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

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

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

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

Consider 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.

Type-Based Reasoning for Implicit Flows in $\lambda^{\star}_{\mathsf{IFC}}$

```
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
```

$\Rightarrow$

```
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 ⟨ ↑ ⟩)
```

$\longrightarrow^*$ (if (1 ⟨ low ! ⟩ < 5 ⟨ ↑ ; high ! ⟩) then 1 ⟨ low ! ⟩ else ...) ⟨ low ?^{p} ⟩

$\longrightarrow^*$ (if (true ⟨ ↑ ; high ! ⟩) then 1 ⟨ low ! ⟩ else ...) ⟨ low ?^{p} ⟩

$\longrightarrow^*$ (prot high (1 ⟨ low ! ⟩)) ⟨ low ?^{p} ⟩

$\longrightarrow^*$ 1 ⟨ ↑ ; high ! ⟩ ⟨ low ?^{p} ⟩

$\longrightarrow^*$ blame p

Road Map

- Background
- $\lambda^{\star}_{\texttt{IFC}}$: a gradual IFC calculus
- ☞ Technical development
  - Formal definition of $\lambda^{\star}_{\texttt{IFC}}$
  - Coercion calculi for IFC
  - The IFC cast calculus $\lambda^{c}_{\texttt{IFC}}$
- Meta-theoretic results

Syntax of $\lambda^{\star}_{\mathtt{IFC}}$

We define a gradual IFC calculus $\lambda^{\star}_{\mathtt{IFC}}$ with mutable references, first-class functions, and conditionals. Highlighted security labels default to `low` if omitted:

$$
\begin{array}{rcl}
\ell & \in & \{\mathtt{low}, \mathtt{high}\} \\
g & \in & \{\mathtt{low}, \mathtt{high}, \star\} \\
\iota & ::= & \mathtt{Unit} \mid \mathtt{Bool} \\
T & ::= & \iota \mid A \xrightarrow{g} A \mid \mathtt{Ref}\ (T_g) \\
A & ::= & T_g \\
M & ::= & x \mid k_{\ \ell} \mid (\lambda^{g} x{:}A.\ M)_{\ \ell} \mid (M\ M)^{p} \\
 & \mid & (\mathtt{if}\ M\ \mathtt{then}\ M\ \mathtt{else}\ M)^{p} \\
 & \mid & (\mathtt{ref}\ \ell\ M)^{p} \mid\ !^{p}\ M \mid (M := M)^{p}
\end{array}
$$

Semantics of $\lambda^{\star}_{\mathsf{IFC}}$

The semantics of $\lambda^{\star}_{\mathsf{IFC}}$ is by translation $\mathcal{C}$ to a cast calculus.

$$\text{evaluation result} \quad r \quad ::= \quad k \mid \mathtt{fun} \mid \mathtt{addr} \mid \mathtt{diverge} \mid \mathtt{stuck}$$

$\boxed{\mathit{obs}(V) = r}$

$$\begin{aligned}
\mathit{obs}(k) &= k \\
\mathit{obs}(k\,\langle\, \boldsymbol{c} \,\rangle) &= k \\
\mathit{obs}(\lambda x.\, N) &= \mathtt{fun} \\
\mathit{obs}((\lambda x.\, N)\,\langle\, \boldsymbol{c} \,\rangle) &= \mathtt{fun} \\
\mathit{obs}(\mathtt{addr}\ n) &= \mathtt{addr} \\
\mathit{obs}((\mathtt{addr}\ n)\,\langle\, \boldsymbol{c} \,\rangle) &= \mathtt{addr}
\end{aligned}$$

Let M be a well-typed $\lambda^{\star}_{\mathsf{IFC}}$ term: $(x{:}\mathtt{Bool}_{\mathtt{high}}); \mathtt{low} \vdash M : \mathtt{Bool}_{\mathtt{low}}$

$\boxed{\mathit{eval}(M, b) = r}$

$$\begin{aligned}
\mathit{eval}(M, b) &= \mathit{obs}(V) && \text{if } (\mathcal{C}\ M)[x := b] \mid \varnothing \Downarrow V \mid \mu \\
\mathit{eval}(M, b) &= \mathtt{diverge} && \text{if } (\mathcal{C}\ M)[x := b] \mid \varnothing \Downarrow \mathtt{blame}\ p \mid \mu \\
& && \text{or } (\mathcal{C}\ M)[x := b] \mid \varnothing \Uparrow \\
\mathit{eval}(M, b) &= \mathtt{stuck} && \text{otherwise}
\end{aligned}$$

The Cast Calculus $\lambda^{c}_{\mathtt{IFC}}$

The casts in $\lambda^{c}_{\mathtt{IFC}}$ are represented by coercions on types (a la Henglein) and coercions on security labels.

$$
\begin{array}{rcl}
c & ::= & \mathrm{id}(g) \mid \uparrow \mid \ell\,\mathtt{!} \mid \ell\,\mathtt{?}^{p} \mid \bot^{p} \\
\bar{c} & ::= & \mathrm{id}(g) \mid \bot^{p} \mid \bar{c}\,\mathtt{;}\,c \\
e & ::= & \ell \mid \mathtt{blame}\ p \mid e\langle \bar{c} \rangle \\
c_r & ::= & \mathrm{id}(\iota) \mid \mathrm{Ref}\ \boldsymbol{c}\ \boldsymbol{c} \mid (\bar{c}, \boldsymbol{c} \to \boldsymbol{c}) \\
\boldsymbol{c} & ::= & (c_r, \bar{c}) \\
M & ::= & x \mid k \mid \lambda x.\, M \mid \mathtt{let}\ x{=}M\mathtt{:}A\ \mathtt{in}\ M \\
 & \mid & M\langle \boldsymbol{c} \rangle \\
 & \mid & M \cdot M \mid M \cdot^{\star} M \\
 & \mid & \mathtt{if}\ M\ \mathtt{then}\ M\ \mathtt{else}\ M \mid \mathtt{if}^{\star}\ M\ \mathtt{then}\ M\ \mathtt{else}\ M \\
 & \mid & \mathtt{ref}\ \ell\ M \mid \mathtt{ref?}^{p}\ \ell\ M \mid\ \mathtt{!}\ M \mid\ \mathtt{!}^{\star}\ M \\
 & \mid & M\ \mathtt{:=}\ M \mid M\ \mathtt{:=?}^{p}\ M \\
 & \mid & \mathtt{addr}\ n \mid \mathtt{prot}\ e\ \ell\ M\ A \mid \mathtt{blame}\ p
\end{array}
$$

Road Map

- Background
- $\lambda^{\star}_{\texttt{IFC}}$: a gradual IFC calculus
- Technical development
- ☞ Meta-theoretic results
  - Type safety for $\lambda^{\star}_{\texttt{IFC}}$
  - Gradual guarantee for $\lambda^{\star}_{\texttt{IFC}}$
  - Noninterference for $\lambda^{\star}_{\texttt{IFC}}$

$\boxed{\vdash r : A}$

$$\frac{k : \iota}{\vdash k : \iota_\ell} \qquad \frac{}{\vdash \mathsf{addr} : (\mathsf{Ref}\ A)_g}$$

$$\frac{}{\vdash \mathsf{fun} : (A \xrightarrow{g_2} B)_{g_1}} \qquad \frac{}{\vdash \mathsf{diverge} : A}$$

Theorem (Type safety of $\lambda^\star_{\mathsf{IFC}}$)

If M *is a well-typed* $\lambda^\star_{\mathsf{IFC}}$ *term:* $(x{:}\mathsf{Bool}_{\mathsf{high}}); \mathtt{low} \vdash M : \mathsf{Bool}_{\mathtt{low}}$
and $\mathit{eval}(M, b) = r$
then the evaluation result is well-typed $\vdash r : \mathsf{Bool}_{\mathtt{low}}$.

Theorem (Gradual guarantee for $\lambda^{\star}_{\mathtt{IFC}}$)

Suppose M and M' are well-typed $\lambda^{\star}_{\mathtt{IFC}}$ terms:

$$(x{:}\mathtt{Bool}_{\mathtt{high}}); \mathtt{low} \vdash M : \mathtt{Bool}_{\mathtt{low}}$$
$$(x{:}\mathtt{Bool}_{\mathtt{high}}); \mathtt{low} \vdash M' : \mathtt{Bool}_{\mathtt{low}}$$

and they are related by precision: $\vdash M \sqsubseteq M'$. If

$$\mathit{eval}(M', b_1) = b_2$$

then

$$\mathit{eval}(M, b_1) = b_2$$

Theorem (Noninterference for $\lambda^{\star}_{\mathtt{IFC}}$)

If M *is a well-typed* $\lambda^{\star}_{\mathtt{IFC}}$ *term:* $(x{:}\mathtt{Bool}_{\mathtt{high}}); \mathtt{low} \vdash M : \mathtt{Bool}_{\mathtt{low}}$
and

$$\mathit{eval}(M, b_1) = b_1'$$

and

$$\mathit{eval}(M, b_2) = b_2'$$

then $b_1' = b_2'$.

Conclusion

- It is possible to satisfy noninterference and the gradual guarantee while supporting type-based reasoning.
- The security labels on constants and memory locations should default to `low` or `high` so that $\star$ is not included in runtime security labels.
- Security checks can be modeled using coercions.

Show me the code! The Agda mechanization of $\lambda^{\star}_{\mathtt{IFC}}$ is at

`https://github.com/Gradual-Typing/LambdaIFCStar`

Thank you! ☺

NSU Checking

$$\boxed{M \mid \mu \mid PC \longrightarrow N \mid \mu'}$$

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

$$\frac{\begin{array}{c}\text{NF } \bar{c} \qquad \vdash \boldsymbol{c} : T_g \Rightarrow S_{\hat{\ell}} \qquad \vdash \boldsymbol{d} : S_{\hat{\ell}} \Rightarrow T_g \\ (\mathit{stamp}!\ PC\ |\bar{c}|) \langle \star \Rightarrow^{p} \hat{\ell} \rangle \longrightarrow^{*} PC' \\ V \langle \boldsymbol{c} \rangle \longrightarrow^{*} W\end{array}}{(\texttt{addr}\ n \langle \text{Ref } \boldsymbol{c}\ \boldsymbol{d}, \bar{c} \rangle)\ \texttt{:=}^{p}\ V \mid \mu \mid PC \longrightarrow \texttt{unit} \mid [\hat{\ell} \mapsto n \mapsto W]\ \mu}$$

Simulation Between More and Less Precise Coercions

Consider the following $\lambda^{\star}_{\mathtt{IFC}}$ terms related by precision:

$$\mathtt{true}_{\mathtt{low}} : \mathtt{Bool}_{\star} : \mathtt{Bool}_{\star} \qquad \text{and} \qquad \mathtt{true}_{\mathtt{low}} : \mathtt{Bool}_{\mathtt{high}} : \mathtt{Bool}_{\star}$$

We need to show the two coercion sequences are related:

$$\vdash \mathrm{id}(\mathtt{low}) \,;\, \mathtt{low}\,! \sqsubseteq \mathrm{id}(\mathtt{low}) \,;\, \uparrow \,;\, \mathtt{high}\,!$$