Mechanized Type Safety for Gradual Information Flow

Tianyu Chen Jeremy G. Siek

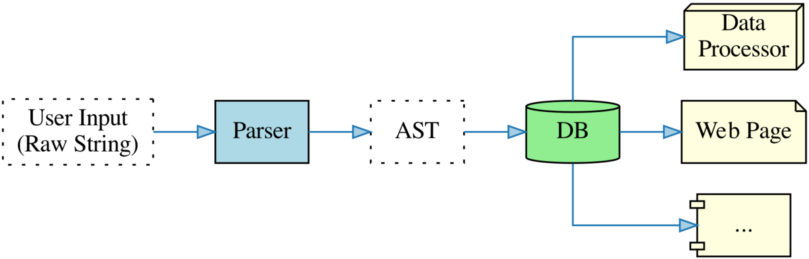
Indiana University

Mosaic pattern, Samarkand. *Wikimedia Image Archive*

Road Map

- Why Gradual Information Flow Typing? ☜
- Interpreting *GLIO*, in *Agda*
- Proving Type Safety
- Existing Designs and Future Directions

The Problem: Protecting Sensitive User Input



Consider an application:

- A user enters a string as input. Selected parts of the string are sensitive.
- Sensitive information in the input string *must not* be disclosed on the web page.

A Solution: *Information Flow Typing*!

- A solution is to implement the application in a *security-typed language*.
  - The language regulates the flow of information and enforce *confidentiality*.
  - It satisfies the *noninterference* property, so sensitive *input* does not interfere with publicly observable *output*: `High` $\not\Rightarrow$ `Low`.
- Implementing the application in a security-typed language will guarantee the *confidentiality* of sensitive user information.

Defining the Grammar with Security Labels

Consider the following user input string:

`{FirstName=Mad;LastName=Hatter;SSN=012-34-5678}`

Defining the Grammar with Security Labels

Consider the following user input string:

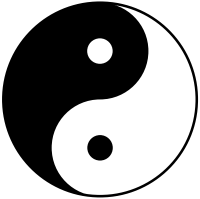
`{FirstName=Mad;LastName=Hatter;SSN=012-34-5678}`

The input grammar:

- High-security terminals are in red; low-security ones in blue.
- The labels are propagated into further processing.
- Thanks to the language satisfying *noninterference*, high-security SSN digits will never be disclosed.

$$
\begin{aligned}
\langle RECORD \rangle &::= \textbf{\{FirstName=}\langle ID \rangle\textbf{;}\\
&\quad\ \textbf{LastName=}\langle ID \rangle\textbf{;}\\
&\quad\ \textbf{SSN=}\langle SSN \rangle\textbf{\}}\\
\langle ID \rangle &::= w, w \in \{\textbf{A}, \ldots \textbf{Z}, \textbf{a}, \ldots \textbf{z}\}^{+}\\
\langle SSN \rangle &::= \langle D \rangle\langle D \rangle\langle D \rangle\textbf{-}\langle D \rangle\langle D \rangle\textbf{-}\langle D \rangle\langle D \rangle\langle D \rangle\langle D \rangle\\
\langle D \rangle &::= d, d \in \{\mathbf{0}, \ldots \mathbf{9}\}
\end{aligned}
$$

Why Gradual Typing?



Static Dynamic

Developer can choose between:

- ✓ Putting in effort to make the program type check at *compile time*.
- ✓ Leaving out the annotations to defer the enforcement until *runtime*.

Our work is based upon *GLIO*, a gradual security-typed language first introduced by de Amorim et al.

A. A. de Amorim, M. Fredrikson, and L. Jia, "Reconciling noninterference and gradual typing", LICS, July 2020.

Road Map

- Why Gradual Information Flow Typing?
- Interpreting *GLIO*, in *Agda* ☜
- Proving Type Safety
- Existing Designs and Future Directions

Enforcing IFC in *GLIO*, Fully Dynamically

Example M^d :

```
let f = λ x : (Lab ¿ Bool) . publish x in
let g = λ x : (Lab ¿ Bool) . (f x) in
let v = to-label High true in
  g v
```

Consider the example program above, M^d:

- The function `publish` publishes a low-security value to publicly visible output.
- Function f, g both take boolean with *statically unknown label* ¿.
- Variable v is bound to a value of high-security.

Enforcing IFC in *GLIO*, Fully Dynamically

$\boxed{\text{Example } M^d :}$

```
let f = λ x : (Lab ¿ Bool) . publish x in
let g = λ x : (Lab ¿ Bool) . (f x) in
let v = to-label High true in
  g v
```

Consider the example program above, M^d:

- The function `publish` publishes a low -security value to publicly visible output.
- Function f, g both take boolean with *statically unknown label* ¿.
- Variable v is bound to a value of high -security.

✓ M^d is well-typed.

Enforcing IFC in *GLIO*, Fully Dynamically

Example M^d :

```
let f = λ x : (Lab ¿ Bool) . publish x in
let g = λ x : (Lab ¿ Bool) . (f x) in
let v = to-label High true in
  g v
```

Consider the example program above, M^d:

- The function `publish` publishes a low -security value to publicly visible output.
- Function f, g both take boolean with *statically unknown label* ¿ .
- Variable v is bound to a value of high -security.

✓ M^d is well-typed.

! Now lets see what happens when we run M^d!

Security Checking at *Runtime*

```
let f = λ x : (Lab ¿ Bool) . publish x in

let g = λ x : (Lab ¿ Bool) . (f x) in

let v = to-label High true in
  g v
```

The implicit casts serve as security checks and catch information flow violation at runtime.

- 1st cast: `Lab High Bool` ⇒ `Lab ¿ Bool` - permitted ✓

Security Checking at *Runtime*

```
let f = λ x : (Lab ¿ Bool) . publish x in

let g = λ x : (Lab ¿ Bool) . (f x) in

let v = to-label High true in
  g v
```

The implicit casts serve as security checks and catch information flow violation at runtime.

- 1st cast: `Lab High Bool` ⇒ `Lab ¿ Bool` - permitted ✓
- 2nd cast: `Lab ¿ Bool` ⇒ `Lab ¿ Bool` - permitted ✓

Security Checking at *Runtime*

```
let f = λ x : (Lab ¿ Bool) . publish x in

let g = λ x : (Lab ¿ Bool) . (f x) in

let v = to-label High true in
  g v
```

The implicit casts serve as security checks and catch information flow violation at runtime.

- 1st cast: `Lab High Bool` ⇒ `Lab ¿ Bool` - permitted ✓
- 2nd cast: `Lab ¿ Bool` ⇒ `Lab ¿ Bool` - permitted ✓
- 3rd cast: `Lab ¿ Bool` ⇒ `Lab Low Bool` - rejected ✗. Execution is terminated due to `castError` and thus information leakage is prevented.

Enforcing IFC in *GLIO*, Fully Statically

Example M^s :

```
let f = λ x : (Lab Low Bool) . publish x in
let g = λ x : (Lab Low Bool) . (f x) in
let v = to-label High true in
   g v ✗
```

- M^s is the fully statically typed counterpart of M^d.
- We annotation f and g with static labels `Low`.
- The program is rejected by the type checker because `High` $\not\preccurlyeq$ `Low`. Security is enforced statically.

Partially Annotated Program

Example M :

```
let f = λ x : (Lab Low Bool) . publish x in
let g = λ x : (Lab ¿ Bool) . (f x) in
let v = to-label High true in
  g v
```

The program M is partially annotated, f has a static annotation while g does *not*.

- The program is statically well-typed, unlike M^s.
- But compared to M^d, the security violation can be detected earlier! We shall see on the next slide when M runs.

Detecting Security Violation in M

```
let f = λ x : (Lab Low Bool) . publish x in

let g = λ x : (Lab ¿ Bool) . (f x) in

let v = to-label High true in
  g v
```

Similar to M^d, security is enforced at runtime:

- 1st cast: `Lab High Bool` $\Rightarrow$ `Lab ¿ Bool` - permitted ✓

Detecting Security Violation in M

```
let f = λ x : (Lab Low Bool) . publish x in

let g = λ x : (Lab ¿ Bool) . (f x) in

let v = to-label High true in
  g v
```

Similar to M^d, security is enforced at runtime:

- 1st cast: `Lab High Bool` ⇒ `Lab ¿ Bool` - permitted ✓
- 2nd cast: `Lab ¿ Bool` ⇒ `Lab Low Bool` - rejected ✗.
  Execution is terminated due to `castError` earlier than M^d because the program is more annotated.

The Definitional Interpreter $\mathcal{V}$

$$\boldsymbol{\mathcal{V}} : \forall\, \Gamma\ T\ \hat{\ell}_1\ \hat{\ell}_2\ .\ (\gamma : \texttt{Env}) \ \to\ (M : \texttt{Term}) \\ \to (\mu : \texttt{Store}) \ \to\ (\texttt{pc} : \mathcal{L}) \ \to\ (k : \mathbb{N}) \\ \to \texttt{Result Conf}$$

$\boxed{\gamma : \texttt{Env}}$ Maps variables to values.

$\boxed{M : \texttt{Term}}$ Runs on a well-typed term.

$\boxed{\mu : \texttt{Store}}$ Maps addresses to type-value pairs $\langle T, v \rangle$.

$\boxed{\texttt{pc} : \mathcal{L}}$ The program counter label to start with.

$\boxed{k : \mathbb{N}}$ Gas; so that the interpreter is total.

$\boxed{\texttt{Result Conf}}$ The evaluation result configuration.

An Example Evaluation (of M)

As expected, evaluating M yields a cast error at runtime:

$\boxed{\text{Evaluating } M:}$

```
run-M : 𝒱 [] M [] Low 42 ≡ error castError
run-M = refl
```

Road Map

- Why Gradual Information Flow Typing?
- Interpreting *GLIO*, in *Agda*
- Proving Type Safety ☜
- Existing Designs and Future Directions

Desirable Language Properties

- de Amorim et al. prove that :
  - **Noninterference:** *GLIO* is secure.
  - **Gradual Guarantees:** Removing annotations, the term remains well-typed and has the same runtime behavior.
- We prove that:
  - **Type Safety:** Undefined behavior never occurs in *GLIO*.
- Future work:
  - **Blame theorem:** A cast cannot be blamed if its source type and target type satisfy subtyping.
  - **Space Efficiency:** Casts are compressed so they do *not* grow in an unbounded fashion.

Why Care About *Type Safety*?

- Distinguish between different types of errors:

  All errors $\begin{cases} \textit{Trapped} \\ \textit{Untrapped} \end{cases}$

- *Untrapped errors* are bad because they are undefined behavior; can be used to hack a program.
- **Type safety:** *untrapped errors* never occur!

Why Care About *Type Safety*?

- Distinguish between different types of errors:

  $$\text{All errors} \begin{cases} \textit{Trapped} \\ \textit{Untrapped} \end{cases}$$

- *Untrapped errors* are bad because they are undefined behavior; can be used to hack a program.
- **Type safety:** *untrapped errors* never occur!
- Machine configuration and evaluation result:

$$\begin{aligned} \mathcal{L} &= \{\texttt{Low}, \texttt{High}\} \\ c &\in \texttt{Store} \times \texttt{Value} \times \mathcal{L}, e \in \texttt{Error} \\ \texttt{Result} &::= \texttt{timeout} | \texttt{error}\ e | \texttt{conf}\ c \end{aligned}$$

Theorem Statement of Type Safety

Theorem (Type safety)

If term M is well-typed: $[] \vdash_{\hat{\ell}_1, \hat{\ell}_2} M : T$, the evaluation result of M is also well-typed:

$$\vdash \mathcal{V} \, [] \, M \, \_ \, [] \, \mathtt{pc} \, k : T$$

Untrapped error is ruled out by well-typedness:

$$\frac{}{\vdash \mathtt{timeout} : T} \qquad \frac{e \in \mathit{TrappedErrors}}{\vdash \mathtt{error}\, e : T}$$

$$\frac{\mu \vdash \mu \qquad \mu \vdash v : T}{\vdash \mathtt{conf}\ \langle \mu, v, \mathtt{pc} \rangle : T}$$

Road Map

- Why Gradual Information Flow Typing?
- Interpreting *GLIO*, in *Agda*
- Proving Type Safety
- Existing Designs and Future Directions ☜

Gradual Security-Typed Language Properties

System	Noninter-ference	Type Safety	Gradual guarantees	Blame theorem	Space efficiency
λ_{gif}	✓ Yes	✓ Yes	✲ Maybe	✓ Yes	✗ No
ML-GS	✓ Yes	✓ Yes	✲ Maybe	✲ Maybe	✗ No
GSL_{Ref}	✓ Yes	✓ Yes	✗ No	✗ No	✲ Maybe
GLIO	✓ Yes	✓ Yes	✓ Yes	✗ No	✗ No

- Two languages that satisfy the most properties are λ_{gif} and *GLIO*. However, as is mentioned earlier, λ_{gif} lacks mutable reference.
- The paper de Amorim et al. proves both *noninterference* and *gradual guarantees* for *GLIO*, resolving the tension proposed in the GSL_{Ref} paper by having casts *checking* labels only, with *classifying* the data.
- Unfortunately, *GLIO* does *not* perform blame tracking. It would be difficult to add blame tracking to *GLIO* due to its heap model.

✲ We summarize our vision for a future design on the next slide.

Language Design Choices & Future Directions

To facilitate *all* five properties, we recommend the following design choices:

- **Value labeling:** Associating values with *concrete* labels (`Low`, `High`, ...); similar to *GLIO*.
- **Heap model:** Simple heap (no extra information stored) and reference proxies.
- **Surface language and cast insertion:** Having both surface language and cast insertion; similar to GSL_{Ref}.
- **Labeling granularity:** Fine-grained labeling; similar to λ_{gif}, *ML-GS*, and GSL_{Ref}.

Thank you!!

Any questions?