Mechanized Type Safety for Gradual Information Flow

Tianyu Chen
Computer Science
Indiana University
Bloomington, USA
chen512@iu.edu

Jeremy G. Siek
Computer Science
Indiana University
Bloomington, USA
jsiek@indiana.edu

***Abstract*—We model a security-typed language with gradual information flow labels in a proof assistant, demonstrate its potential application to parsing and securing sensitive user input data, present the semantics as a definitional interpreter, and prove type safety. We compare the language features and properties of various existing gradual security-typed languages, shedding light on future designs.**

***Index Terms*—gradual typing, information flow security, mechanized metatheory**

I. Introduction

In this paper, we prove type safety for a language with *gradual* information flow labels. That is, a language in which the programmer can request that information flow be checked statically or they can defer such checking until *runtime* by using the unknown information flow label, written ¿, in type annotations. We describe how information-flow secure language could be applied to constructing a parser that protects sensitive user input. Specifically, we focus on *GLIO*, a language introduced by de Amorim et al. [1], who prove that it satisfies *noninterference* [1, section 5] and the *gradual guarantees* [1, section 6]. However, there are several more properties that one expects of such a language: type safety, blame safety, conservativity, and dynamic embedding [2]. In this paper we focus on the first of those properties, type safety. Unfortunately, the denotational semantics of GLIO given by de Amorim et al. [1] does not distinguish between trapped and untrapped errors, which makes it impossible to formulate the type safety property. To remedy this we present a definitional interpreter for *GLIO* that distinguishes between two kinds of trapped errors (failure of a *no-sensitive-upgrade* (NSU) check or a cast from high to low security) and the other errors which are untrapped. Our type safety proof, as usual, guarantees that untrapped errors never occur. The definitional interpreter and the proof of type safety are mechanized in the *Agda* proof assistant.

Broadly speaking, we are interested in *confidentiality*, that is, restricting information access to authorized parties only, which forms a triad together with integrity and availability as the foundation of information security [3]. From the perspective of a programming language, confidentiality is often formalized as satisfying *noninterference*, a theorem stating that high-security input must not affect publicly observable low-security outputs [4].

Modern software applications often accept user input where selected fields are sensitive, whose confidentiality is required during both parsing and processing. Suppose we have a web application that receives three fields from its user: 1) first name 2) last name 3) social security number, the grammar of which is defined in figure 1, where terminals are divided into low-security and high-security. The digits d for social security number, being confidential to users of the web application, are of high-security, so they are marked red, while other terminals, such as the keys of the record and the strings w for first name / last name, being safe to disclose, are all of low-security, marked in blue.

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

Fig. 1: Example grammar for user input

Consider the following user input:
`{FirstName=Mad;LastName=Hatter;SSN=012-34-5678}`

The author of the web application could implement a parser for the grammar in Figure 1 in a language that enforces information flow security. Each terminal in the grammar would be labeled with a security level and the language would guarantee that the high-security information is only present in those parts of the output parse tree that are marked as high-security. For example, according to the grammar in figure 1, the example user input string is parsed into the parse tree in figure 2, where the terminal nodes that represent digits of the social security number are of high-security, while the terminals that compose the rest of the input string are of low-security. The confidentiality of SSN is guaranteed during data processing since the language enforces noninterference. When the web application interacts with the outside world, such as making a foreign function interface (*FFI*) call or storing into

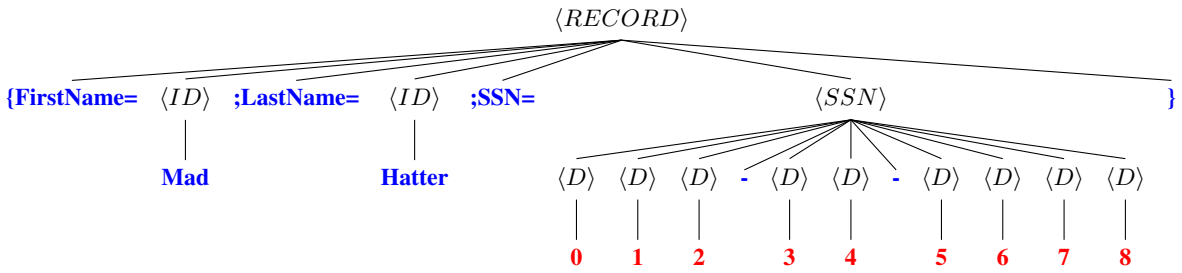

Fig. 2: The parse tree generated from the example user input. All terminals are represented as labeled values: the red ones, such as the digits of SSN, are of high-security, while the blue ones, such as the keys of the record and first name / last name, are of low-security.

a database, the conceptual language need to encrypt whatever values labeled as high security before they are passed into a foreign routine.

The interest in enforcing confidentiality and regulating the flow of information in a computer program arises with its defense applications in the 1970s [5]. Denning [6] builds a information flow model using a lattice of security labels and Denning and Denning [7] discuss the certification technique in further detail, with a proof that a certified program will not give away confidential input from non-confidential output. Volpano et al. [8] propose a typed-based approach to enforcing information flow, by defining a type system for an imperative programming language and proving its security with a type soundness proof. This idea is further developed by Zdancewic and Myers [9]. Such a protection scales well since type checking is *compositional* [10]. There are projects that apply similar techniques but to other languages such as bytecode intermediate languages [11], object-oriented languages [12], and reactive programming languages [13]. Although the aforementioned languages are mostly theoretical, efforts have also been made to integrate information flow control into widely-used existing languages such as *Jif* for Java [14] and *Flow Caml* for OCaml [15, 16].

While type systems rule out undesired flows statically, it is also possible to do so at runtime. Li and Zdancewic [17] add flow control to Haskell by utilizing existing language features, specifically *arrow* and *typeclass*, to implement checks. Similarly, the *LIO* library employs a labeled IO monad to keep track of the current privilege level, which restricts both the observability and the security effect [18, 19]. *LIO* also provides 1) first-class labels so that labels are values and can be manipulated by the programmer on-the-fly, and 2) coarse-grained labeling so that a programmer may choose to label a value when it is necessary to impose flow control policies and omit the labels for security-insensitive parts of the program.

The *HLIO* [20] library introduces *hybrid* checking of information flow, whereby a programmer can choose between static or dynamic checking in different parts of the program. By default the checking is static, but a programmer can insert a `defer` clause to say that the security constraints should be checked at runtime.

Gradual typing [21, 22, 23] is a paradigm that combines static typing and dynamic typing - checks are performed at the boundaries between statically and dynamically typed code fragments. The most obvious difference from static typing is that a gradual type system usually contains a *dynamic* type that stands for the type that is statically unknown - in our case, it is a dynamic information flow label written ¿. Unlike the hybrid approach discussed above, the programmer does not insert casts or `defer` clauses to mark the transitions between static and dynamic.

Recently there has been increasing interest in building gradual security-typed languages. The benefit of having a gradual security-typed language is that the programmer can choose when it is appropriate to put in the effort to pass the static security checks and when it is appropriate to defer such enforcement to runtime - where information flow violations will appear as trapped errors. Gradual typing facilitates migration between the static and dynamic checking because, roughly speaking, changes in type annotations are guaranteed to preserve program behavior, a property called the gradual guarantee [2].

GLIO [1], on which this paper is based, is comparable to *HLIO* in that both enable deferring information flow checks to runtime and have similar features to the *LIO* language mentioned above. They are different in that the latter is *gradual* instead of *hybrid*, checks are guided by *type* and a developer does not need to embed explicit casts into the program. Additionally, *GLIO* satisfies criteria of a gradually-typed language, such as dynamic and static gradual guarantees, making the migration between paradigms easy. Apart from *GLIO*, there are a few other noteworthy designs: Disney and Flanagan [24] explore the idea of adding explicit casts to enable dynamic flow control for a purely functional security-typed language called λ_{gif}. Fennell and Thiemann [25] present a similar cast calculus, *ML-GS*, with mutable references, a feature that λ_{gif} lacks. The heap of *ML-GS* stores both values and types, a model that *GLIO* follows. Toro et al. [26] derive a language, GSL_{Ref}, leveraging the *AGT framework* [27]. GSL_{Ref} contains both a surface language and a cast calculus. NSU checks are derived from their static counterparts during the cast-insertion (called elaboration in GSL_{Ref}) procedure. The authors also show that the typing rules of GSL_{Ref} stay relatively similar to its fully static version SSL_{Ref}, other than a few relations being replaced by their gradual counterparts.

With the growth in gradual security-typed language designs,

we find it beneficial to make a thorough comparison between them. We are interested in their language features and properties. Additionally, their heap models are also worth studying, since a few of the designs store additional information beside a value at each memory cell.

To summarize, the contributions of this work are:

- A definitional interpreter for *GLIO* that distinguishes between different types of trapped errors.
- We prove type safety using the aforementioned semantics: a well-typed *GLIO* program never gets stuck and always evaluates to a well-typed machine configuration.
- We compare various existing gradual security-typed programming languages and discuss their design choices. We provide insight into which criteria future designs should satisfy.

II. Review of the *GLIO* Language

In this section, we briefly review the syntax of *GLIO* [1] and the type system in the form of *Agda* code [1]. *Agda* is a dependently-typed programming language and proof assistant [28]. The code and machine-checked proofs in this paper are available at https://github.com/Gradual-Typing/lambda-sec/tree/master/glio.

A. GLIO by Example

GLIO is a gradually-typed language. Gaps between the statically-typed fragments and the dynamically-typed fragments are bridged by implicit casts, which also serve as runtime information flow checks. Consider the program in listing 1 that demonstrates how a function g, whose parameter has a dynamic information flow label, interacts with a statically typed region of code. Suppose there are two security levels: `Low` and `High`. The type annotation on each let binding is omitted and defaults to the type of the expression on the right-hand side. The program counter annotation of each λ-abstraction defaults to `Low`. A value with type T protected by a certain information flow label ℓ inhabits `Lab` ℓ `T`; the label may optionally be determined at runtime, written `Lab ¿ T`, where the ¿ means a statically unknown label.

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

Listing 1: Example of casts

Listing 1 shows a well-typed program. However, it may leak information, since function `f` publishes a low-security variable, while the value passed through function `g`, whose parameter is of a statically unknown security level, is a high-security boolean. To ensure security, *GLIO* terminates the execution due to a failed cast and prevents the high-security value from being disclosed. At the application `g v`, the boolean labeled `High` that variable `v` is bound to is cast from `Lab High Bool` to `Lab ¿ Bool`, which is permitted. The application `f x` in the body of `g`, on the other hand, attempts to cast a high-security boolean from `Lab ¿ Bool` to `Lab Low Bool`, which errors due to a failed runtime check.

If every annotation is decorated with a concrete label, *GLIO* can discover illegal information flows during type checking, just like a statically security-typed language. Consider the following program, `Low` and `High`:

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

Listing 2: Example of a statically-typed program, rejected

This program is unsafe since a high-security boolean is passed into the identity function `f` which takes a low-security boolean, ruled out by the type system. To make the program type check, the programmer may lower the label on `v` like in a statically security-typed language:

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

Listing 3: Example of a statically-typed program, fixed

GLIO is a coarse-grained security-typed language, which means that not all types and values are labeled. Values of unlabeled types default to publicly visible. It also provides first-class labels (labels as values). Consider the following variation of the examples in listings 2 and 3:

```
let ℓ = (user-input) in
  let f = λ x : (Lab Low Bool) . x in
    let v = to-label-dyn ℓ true in
      f v
```

Listing 4: Example of dynamically labeled value

There are two changes in the example above compared with its static variants: 1) The `to-label` clause is replaced with `to-label-dyn`. 2) A variable ℓ, bound to the user's input, is provided instead of a concrete label. If the user input label is `High` when the program executes, a check happens during the application, casting a boolean value labeled as high-security to the type `Lab Low Bool`, which fails and triggers a cast error, thus preventing the program from leaking information. If the user input is `Low` instead, the program finishes successfully and returns a boolean of low-security.

Similarly, consider the following example, where a new heap location is created to store a secret:

```
let x = to-label High true in
  let y = unlabel x in
    new Low y
```

Listing 5: Creating labeled reference, statically rejected

The boolean is labeled as `High`, unlabeled and then written to a new heap location when security level `Low`. It contains an unsafe information flow from high-security to low-security and is ruled out by the type checker. We may assign the secrecy of the newly created heap cell at runtime:

[1] *Agda* 2.6.1.2 and standard library 1.4

```
let ℓ = (user-input) in
  let x = to-label High true in
    let y = unlabel x in
      new-dyn ℓ y
```

Listing 6: Creating dynamically labeled reference

If the input label ℓ is `High` when the program runs, it finishes successfully since the information is flowing from high-security to high-security; otherwise if ℓ is `Low`, an NSU error occurs due to a failed runtime check, preventing the program from leaking information. We explain different types of errors in further detail in section III.

B. Defining the Syntax in Agda

The information flow label is defined as a datatype $\mathcal{L}$ with constructor `l` that takes a natural number privilege level:

```
data ℒ : Set where
  l : ℕ → ℒ
```

It forms a lattice with the order $\preccurlyeq$, join $\sqcup$, and meet $\sqcap$, which are analogous to their natural number counterparts $\leq$, $\sqcup_n$, and $\sqcap_n$. We use $\hat{\mathcal{L}}$ for the type of a gradual label - it can be either a concrete label or dynamic ¿:

```
data ℒ^ : Set where
  ¿ : ℒ^
  l^ : ℒ → ℒ^
```

For `Low` and `High`, we have the following shorthand:

```
𝑳 = l 0; 𝑳^ = l^ 𝑳; 𝑯 = l 1; 𝑯^ = l^ 𝑯
```

The definitions of gradual join of labels ($\curlyvee$) and types ($\vee$), gradual meet of labels ($\curlywedge$) and types ($\wedge$), and consistent subtyping of labels ($\precsim$) and types ($\lesssim$) are all defined the same as in de Amorim et al. [1] and thus omitted.

The types are defined using the gradual label datatype $\hat{\mathcal{L}}$:

```
data 𝕋 : Set where
  `⊤ : 𝕋                              -- Unit
  `𝔹 : 𝕋                              -- Bool
  `ℒ : 𝕋                              -- Label
  Ref : ℒ^ → 𝕋 → 𝕋                    -- Reference
  Lab : ℒ^ → 𝕋 → 𝕋                    -- Labeled
  _[_]⇒[_]_ : 𝕋 → ℒ^ → ℒ^ → 𝕋 → 𝕋 -- Function
```

Most cases are straightforward. The label on the reference type denotes the secrecy level of the heap location. There are two labels decorated on a function type, which serve as the static program counters before and after the computation of the body of a λ-abstraction, corresponding to the two labels on the signature of the typing rule (introduced below).

The terms of the *GLIO* are defined in *Agda* using the abstract binding tree library (https://github.com/jsiek/abstract-binding-trees). The terms are extrinsically typed; a term `M` that is typed `T` under typing context Γ and program counter labels $\hat{\ell}_1$ and $\hat{\ell}_2$ is written `Γ [ ℓ^₁ , ℓ^₂ ]⊢ M ⦂ T`. The typing context is defined as a list of types (`List 𝕋`), since we use De Bruijn notation for variables. The two program counter labels $\hat{\ell}_1$ and $\hat{\ell}_2$ are for the security effects *before* and *after* the computation, which restrict heap operations in their respective scopes. We omit the complete definitions of typing rules since they are ported from the *GLIO* paper [1]; A-normal form is used so that the syntax stays close.

III. Interpreting *GLIO*

In this section we present the interpreter for *GLIO*.

A. Modeling the Heap

A heap (`Store`) is defined as a list of cells; each cell maps a location to a type-value pair:

```
data Cell (X : Set) : Set where
  _↦_ : Location → X → Cell X

Store = List (Cell (𝕋 × Value))
```

A location is defined as a 3-tuple consisting of an index number that is unique to each cell, a stored program counter label at allocation, and a security level. It is intentionally kept close to the definition in de Amorim et al. [1]:

```
Location = ℕ × ℒ × ℒ
```

The result of the interpreter is a value, which can be a unit, a boolean, a label, a heap reference, a labeled value, a closure , or a function proxy. A function proxy is a function-typed value wrapped together with the source type, the target type, and the proofs that the source is the consistent subtype of the target:

```
data Value : Set where
  V-tt    : Value
  V-true  : Value
  V-false : Value
  V-label : ℒ → Value
  V-clos  : Clos → Value
  V-proxy : (S T S′ T′ : 𝕋) → (ℓ^₁ ℓ^₂ ℓ^₁′ ℓ^₂′ : ℒ^)
    → S′ ≲ S → S ≲ T → T ≲ T′ → ℓ^₁′ ⋨ ℓ^₁ → ℓ^₂ ⋨ ℓ^₂′
    → Value
    → Value
  V-ref   : Location → Value
  V-lab   : ℒ → Value → Value
```

A closure is a well-typed term with an environment. An environment is a list of values. If there are multiple occurrences of the same address, the heap lookup function picks the first match in the list. Consequently for heap update, we simply *cons* a new type-value pair onto the heap.

B. The Machine Configuration

A *configuration* is defined as a 3-tuple that combines a heap, the resulting value, and a program counter label after the evaluation: `Conf = Store × Value × ℒ`. An execution can either succeed with a valid configuration or fail, by either running out of time or reporting an error:

```
data Result (X : Set) : Set where
  timeout : Result X
  error : Error → Result X
  result : X → Result X
```

The interpreter is defined as a function in *Agda*, which must be total. Any finite execution can be obtained by increasing the argument k [29]. There are three types of errors: cast errors, NSU errors, and untrapped errors:

```
data Error : Set where
  stuck : Error
  castError : Error
  NSUError : Error
```

We define a monadic bind ($\gg=$) to thread the computation, which proceeds if the result from the last step is a valid configuration and aborts otherwise.

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

$$\mathcal{V}\ \_\ \_\ \_\ \_\ 0 \ =\ timeout$$

$$\mathcal{V}\ \gamma\ (`\ x)\ \mu\ pc\ (k+1) \ =\ \begin{cases} result\ \langle \mu, v, pc\rangle & \text{, if } \gamma[x] \equiv v \\ error\ \texttt{stuck} & \text{, if } \gamma[x] \text{ is undefined} \end{cases}$$

$$\mathcal{V}\ \gamma\ (if\ (`\ x)\ M\ N)\ \mu\ pc\ (k+1) \ =\ \begin{cases} do & \\ \quad \langle \mu', v_m, pc'\rangle \ \leftarrow\ \mathcal{V}\ \gamma\ M\ \mu\ pc\ k & \\ \quad \langle \mu'', \_, pc''\rangle \ \leftarrow\ castL\ \mu'\ pc'\ \hat{\ell}_2\ (\hat{\ell}_2 \curlyvee \hat{\ell}'_2) & \\ \quad castT\ \mu''\ pc''\ T\ T''\ v_m & \text{, if } \gamma[x] \equiv \texttt{V}_{\texttt{true}} \\ do & \text{, where } T \vee T' \equiv T'' \\ \quad \langle \mu', v_n, pc'\rangle \ \leftarrow\ \mathcal{V}\ \gamma\ N\ \mu\ pc\ k & \\ \quad \langle \mu'', \_, pc''\rangle \ \leftarrow\ castL\ \mu'\ pc'\ \hat{\ell}'_2\ (\hat{\ell}_2 \curlyvee \hat{\ell}'_2) & \\ \quad castT\ \mu''\ pc''\ T'\ T''\ v_n & \text{, if } \gamma[x] \equiv \texttt{V}_{\texttt{false}} \\ error\ \texttt{stuck} & \text{, otherwise} \end{cases}$$

$$\mathcal{V}\ \gamma\ (\lambda\ N)\ \mu\ pc\ (k+1) \ =\ result\ \langle \mu, \texttt{V}_{\texttt{clos}}\ N\ \gamma, pc\rangle$$

$$\mathcal{V}\ \gamma\ ((`\ x)\ \cdot\ (`\ y))\ \mu\ pc\ (k+1) \ =\ \begin{cases} do & \\ \quad \langle \mu', w', pc'\rangle \ \leftarrow\ castT\ \mu\ pc\ T'\ T\ w & \\ \quad \langle \mu'', \_, pc''\rangle \ \leftarrow\ castL\ \mu'\ pc'\ \hat{\ell}'_1\ \hat{\ell}_1 & \\ \quad apply\ \gamma\ v\ w'\ \mu\ pc\ k & \text{, if } \gamma[x] \equiv v\ ,\ \gamma[y] \equiv w \\ error\ \texttt{stuck} & \text{, otherwise} \end{cases}$$

$$\text{, where } \Gamma[x] \equiv T \xrightarrow{\hat{\ell}_1, \hat{\ell}_2} S,\ \Gamma[y] \equiv T',\ \Gamma \vdash_{\hat{\ell}'_1, \hat{\ell}_2} (`\ x)\ \cdot\ (`\ y) : S$$

$$\boxed{\boldsymbol{apply} : Env \to Value \to Value \to Store \to (pc : \mathcal{L}) \to (k : \mathbb{N}) \to Result\ Conf}$$

$$apply\ \gamma\ (\texttt{V}_{\texttt{clos}}\ N\ \rho)\ w\ \mu\ pc\ k \ =\ \mathcal{V}\ (w :: \rho)\ N\ \mu\ pc\ k$$

$$\begin{aligned} apply\ \gamma\ (\texttt{V}_{\texttt{proxy}}\ S \xrightarrow{\hat{\ell}_1, \hat{\ell}_2} T\ S' \xrightarrow{\hat{\ell}'_1, \hat{\ell}'_2} T'\ v)\ w\ \mu\ pc\ k \ =\ & do \\ & \langle \mu_1, w', pc_1\rangle \ \leftarrow\ castT\ \mu\ pc\ S'\ S\ w \\ & \langle \mu_2, \_, pc_2\rangle \ \leftarrow\ castL\ \mu_1\ pc_1\ \hat{\ell}'_1\ \hat{\ell}_1 \\ & \langle \mu_3, v_1, pc_3\rangle \ \leftarrow\ apply\ \gamma\ v\ w'\ \mu_2\ pc_2\ k \\ & \langle \mu_4, \_, pc_4\rangle \ \leftarrow\ castL\ \mu_3\ pc_3\ \hat{\ell}_2\ \hat{\ell}'_2 \\ & castT\ \mu_4\ pc_4\ T\ T'\ v_1 \end{aligned}$$

$$apply\ \gamma\ \_\ w\ \mu\ pc\ k \ =\ error\ \texttt{stuck}$$

Fig. 3: The definitional interpreter of *GLIO*: timeout, variable, if, abstraction, and application

C. The Definitional Interpreter of GLIO

The interpreter is defined as a total function $\mathcal{V}$ that depends on two helpers, `castL` and `castT`, whose definitions are detailed in appendix A. There are cases where `castT` could possibly get stuck. We show that those cases are never encountered by proving type safety in section IV.

For simplicity, we show only the interesting cases of $\mathcal{V}$, which are split into three parts: abstraction and application (figure 3), heap access (figure 4), and labeling operations (figure 5). The interpreter $\mathcal{V}$ takes an environment γ, a well-typed term M, an original store μ, a program counter pc, and a natural number k called *gas* which makes $\mathcal{V}$ total.

If gas runs out, $\mathcal{V}$ returns a `timeout`, shown in the first case of figure 3. Variable lookup is straightforward, which simply returns the value that variable x corresponds to in the environment γ. If the lookup fails, it means that the term is open and we get stuck. When evaluating `if`, we dispatch on the value of the condition x. We go to the first branch if x is `V-true`, the second if it is `V-false`, and get stuck if it is neither. In either case, the sub-term is evaluated, the label is cast to be the gradual join of the two branches, and the value from the sub-term is then cast to the gradual join of types from both branches.

In λ-abstraction's case, we build a closure by wrapping the well-typed body N and the current environment γ in `V-clos`. Function application is defined leveraging an auxiliary helper `apply` that applies a value which is either a `V-clos` or a `V-proxy` and otherwise gets stuck. If the value is a closure, we directly dive into the body N with an extended environment $w :: \rho$, where ρ is the environment wrapped in the closure. On

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

$$\mathcal{V}\ \gamma\ (get\ (`\ x))\ \mu\ pc\ (k+1)\ =\ \begin{cases} \begin{cases} castT\ \mu\ (pc \sqcup \ell_2)\ T'\ T\ v & \text{, if } \mu[\langle n, \ell_1, \ell_2\rangle] \equiv \langle T', v\rangle \\ error\ \texttt{stuck} & \text{, if } \mu[\langle n, \ell_1, \ell_2\rangle] \text{ is undefined} \end{cases} \\ \quad \text{, if } \gamma[x] \equiv \texttt{V}_{\texttt{ref}}\ \langle n, \ell_1, \ell_2\rangle \\ error\ \texttt{stuck} \quad \text{, otherwise} \end{cases}$$
$$\text{, where } \Gamma[x] \equiv \texttt{Ref}\ \hat{\ell}\ T$$

$$\mathcal{V}\ \gamma\ (set\ (`\ x)\ (`\ y))\ \mu\ pc\ (k+1)\ =\ \begin{cases} \begin{cases} do \\ \quad \langle \mu', v', pc'\rangle\ \leftarrow\ castT\ \mu\ pc\ T'\ T\ v \\ \quad \langle \mu'', v'', pc''\rangle\ \leftarrow\ castT\ \mu'\ pc'\ T\ T''\ v' \\ \quad setmem\ \mu''\ \langle n, \ell_1, \ell_2\rangle\ pc''\ \langle T'', v''\rangle & \text{, if } \mu[\langle n, \ell_1, \ell_2\rangle] \equiv \langle T'', \_\rangle \\ error\ \texttt{stuck} & \text{, if } \mu[\langle n, \ell_1, \ell_2\rangle] \text{ is undefined} \end{cases} \\ \quad \text{, if } \gamma[x] \equiv \texttt{V}_{\texttt{ref}}\ \langle n, \ell_1, \ell_2\rangle \text{ and } \gamma[y] \equiv v \\ error\ \texttt{stuck} \quad \text{, otherwise} \end{cases}$$
$$\text{, where } \Gamma[x] \equiv \texttt{Ref}\ \hat{\ell}\ T\ ,\ \Gamma[y] \equiv T'$$

$$\mathcal{V}\ \gamma\ (new\ \ell\ (`\ y))\ \mu\ pc\ (k+1)\ =\ \begin{cases} \begin{cases} result\ \langle \langle n, pc, \ell\rangle \mapsto \langle T, v\rangle :: \mu, \texttt{V}_{\texttt{ref}}\ \langle n, pc, \ell\rangle, pc\rangle & \text{, where } n \text{ is fresh, if } \gamma[y] \equiv v \\ error\ \texttt{stuck} & \text{, otherwise} \end{cases} \\ \quad \text{, if } pc \preccurlyeq \ell \\ error\ \texttt{NSUError} \quad \text{, otherwise} \end{cases}$$
$$\text{, where } \Gamma[y] \equiv T$$

$$\mathcal{V}\ \gamma\ (new_{dyn}\ (`\ x)\ (`\ y))\ \mu\ pc\ (k+1)\ =\ \begin{cases} \begin{cases} result\ \langle \langle n, pc, \ell\rangle \mapsto \langle T, v\rangle :: \mu, \texttt{V}_{\texttt{ref}}\ \langle n, pc, \ell\rangle, pc\rangle & \text{, where } n \text{ is fresh, if } pc \preccurlyeq \ell \\ error\ \texttt{NSUError} & \text{, otherwise} \end{cases} \\ \quad \text{, if } \gamma[x] \equiv \texttt{V}_{\texttt{label}}\ \ell \text{ and } \gamma[y] \equiv v \\ error\ \texttt{stuck} \quad \text{, otherwise} \end{cases}$$
$$\text{, where } \Gamma[y] \equiv T$$

$$\boxed{\boldsymbol{setmem} : (\mu : Store) \to Location \to (pc : \mathcal{L}) \to \mathbb{T} \times Value \to Result\ Conf}$$

$$setmem\ \mu\ \langle n, \ell_1, \ell_2\rangle\ pc\ tv\ =\ \begin{cases} result\ \langle \langle n, \ell_1, \ell_2\rangle \mapsto tv :: \mu, \texttt{V}_{\texttt{tt}}, pc\rangle & \text{, if } pc \preccurlyeq \ell_2 \\ error\ \texttt{NSUError} & \text{, otherwise} \end{cases}$$

Fig. 4: The definitional interpreter of *GLIO*: heap memory operations

the other hand if the value is a function proxy, we first cast the type of the domain and the program counter at the beginning of the computation, after which we recursively call `apply`. Then we cast the program counter after the computation and the type of the codomain, on the value after the application. In short, applying a proxy unwraps one layer of `V-proxy` into casts. The application case of $\mathcal{V}$ casts the domain on the value w and the program counter and subsequently calls `apply`.

The semantics for heap operations are shown in figure 4, When reading from a heap location $\langle n, \ell_1, \ell_2\rangle$ typed `Ref` $\hat{\ell}$ `T`, we first look it up in μ. If the index is out-of-bound, we run into a memory access error and the evaluation gets stuck. Otherwise if the type-value pair on heap μ is $\langle T', v\rangle$, we cast the value v from T' to T, since we expect a T from the heap reference. When writing to a heap location, we use an auxiliary function `setmem` to check for NSU error, since writing to a location that is less secure than the current program counter must be strictly ruled out. If the index is out-of-bound, we get stuck. Otherwise, we cast the value to store from its original type T' to the type on the reference, T, and then to the type annotation on the heap cell, T''. Finally we invoke `setmem` to write the value to the heap. The operation `new` and `new-dyn` create a new cell on the heap. The difference is whether the label of secrecy ℓ comes statically from the term or dynamically from the environment γ.

Figure 5 shows the cases for `unlabel` and `to-label`. When unlabeling a value, apart from peeling off the label ℓ and retrieving the unwrapped value v, we need to upgrade the program counter by joining with ℓ - this is why the example program in listing 6 may fail at runtime. The cases for `to-label` and `to-label-dyn` are the same except that in the former the label ℓ comes from the term while in the latter ℓ comes from the environment γ - similar to the difference between `new` and `new-dyn`. The two labeling operations both evaluate the sub-term M, perform an NSU check, and return the value v labeled with ℓ.

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

$$\mathcal{V}\ \gamma\ (unlabel\ (`\ x))\ \mu\ pc\ (k+1) = \begin{cases} result\ \langle \mu, v, pc \sqcup \ell \rangle & \text{, if } \gamma[x] \equiv \mathtt{V_{lab}}\ \ell\ v \\ error\ \mathtt{stuck} & \text{, otherwise} \end{cases}$$

$$\mathcal{V}\ \gamma\ (tolabel\ \ell\ M)\ \mu\ pc\ (k+1) = \begin{cases} \begin{cases} result\ \langle \mu', \mathtt{V_{lab}}\ \ell\ v, pc \rangle & \text{, if } pc' \preccurlyeq pc \sqcup \ell \\ error\ \mathtt{NSUError} & \text{, otherwise} \end{cases} \\ \quad \text{, if } \mathcal{V}\ \gamma\ M\ \mu\ pc\ k \equiv result\ \langle \mu', v, pc' \rangle \\ error\ err \quad \text{, if } \mathcal{V}\ \gamma\ M\ \mu\ pc\ k \equiv error\ err \\ timeout \quad \text{, if } \mathcal{V}\ \gamma\ M\ \mu\ pc\ k \equiv timeout \end{cases}$$

$$\mathcal{V}\ \gamma\ (tolabel_{dyn}\ (`\ x)\ M)\ \mu\ pc\ (k+1) = \begin{cases} \begin{cases} \begin{cases} result\ \langle \mu', \mathtt{V_{lab}}\ \ell\ v, pc \rangle & \text{, if } pc' \preccurlyeq pc \sqcup \ell \\ error\ \mathtt{NSUError} & \text{, otherwise} \end{cases} \\ \quad \text{, if } \mathcal{V}\ \gamma\ M\ \mu\ pc\ k \equiv result\ \langle \mu', v, pc' \rangle \\ error\ err \quad \text{, if } \mathcal{V}\ \gamma\ M\ \mu\ pc\ k \equiv error\ err \\ timeout \quad \text{, if } \mathcal{V}\ \gamma\ M\ \mu\ pc\ k \equiv timeout \end{cases} \\ \quad \text{, if } \gamma[x] \equiv \mathtt{V_{label}}\ \ell \\ error\ \mathtt{stuck} \quad \text{, otherwise} \end{cases}$$

Fig. 5: The definitional interpreter of *GLIO*: labeling

Well-typed environment:

$$\frac{}{[]; \mu \vdash []} \qquad \frac{\mu \vdash v : T \quad \Gamma; \mu \vdash \gamma}{T :: \Gamma; \mu \vdash v :: \gamma}$$

Well-typed heap:

$$\frac{}{\mu \vdash []} \qquad \frac{\mu \vdash v : T \quad \mu \vdash \sigma}{\mu \vdash \langle n, \ell_1, \ell_2 \rangle \mapsto \langle T, v \rangle :: \sigma}$$

Well-typed computation result:

$$\frac{\mu \vdash \mu \quad \mu \vdash v : T}{\vdash result\ \langle \mu, v, pc \rangle : T} \qquad \frac{}{\vdash timeout : T}$$

$$\frac{}{\vdash error\ \mathtt{castError} : T} \qquad \frac{}{\vdash error\ \mathtt{NSUError} : T}$$

Fig. 6: Well-typed environment, heap, and computation result

We can see that our interpreter follows a similar structure to the denotational semantics in de Amorim et al. [1, figure. 13], except than the authors use CPO sets to represent the denotation of terms and casts, while we present the semantics as an interpreter that employs a machine configuration monad, which is easier to reason about in a proof assistant, since the mechanized proof simply follows the branches in the interpreter's code. Another benefit is that we can view the evaluation in action - in fact, all the examples shown in section II are runnable in this interpreter.

IV. Mechanized Type Safety Proof in *Agda*

We are now ready to prove type safety with the operational semantics in section III. It is proved by showing the computation result of the interpreter is always well-typed, given well-typed input.

The definitions of well-typedness for environment, heap, computation, and value are given in figure 6 and 7. The typing of environment is quantified by a typing context Γ and a store typing μ. We use the heap *itself* as the store typing context, as we store the type of each cell together with the value. A well-typed environment means that each value in it is well-typed according to its corresponding type in Γ. The typing of value is quantified by the store typing μ, which is necessary due to mutable reference. In the two cases of reference, we only require that the heap location is valid; the type in the cell may not necessarily be the same as the one on the reference. The typing of heap is straightforward; since we store types directly on heap, the type and value need to jive in each cell. Finally, every result may be well-typed *except* stuck - we would like to prove that the program *never* gets stuck. For a configuration $\langle \mu, v, pc \rangle$ to be well-typed, both the heap μ and the value v must be well-typed.

The statement of type safety is in theorem 2, which is a corollary of proposition 1.

Proposition 1 (The interpreter $\mathcal{V}$ is type safe)**.** *If the initial heap is well-typed $\mu \vdash \mu$, the initial environment is well-typed $\Gamma; \mu \vdash \gamma$, and the term is well-typed $[] \vdash_{\hat{\ell}_1, \hat{\ell}_2} M : T$, then the evaluation result is well-typed $\vdash \mathcal{V}\ \gamma\ M\ \_\ \mu\ pc\ k : T$.*

The detailed proof is discussed in appendix B.

Theorem 2 (Type safety)**.** *If term M is well-typed:*

$$[] \vdash_{\hat{\ell}_1, \hat{\ell}_2} M : T$$

, then evaluating M gets a well-typed result:

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

Proof. This theorem is a special case of proposition 1, where the initial environment and heap are both empty, which are trivially well-typed. ■

$$\textit{Unit}\ \frac{}{\mu \vdash \mathtt{V_{tt}} : \text{`}\top} \qquad \textit{Label}\ \frac{}{\mu \vdash \mathtt{V_{label}}\ \ell : \text{`}\mathcal{L}} \qquad \textit{True}\ \frac{}{\mu \vdash \mathtt{V_{true}} : \text{`}\mathbb{B}} \qquad \textit{False}\ \frac{}{\mu \vdash \mathtt{V_{false}} : \text{`}\mathbb{B}}$$

$$\textit{Closure}\ \frac{\Gamma; \mu \vdash \gamma \qquad T :: \Gamma \vdash_{\hat{\ell}_1, \hat{\ell}_2} M : S}{\mu \vdash \mathtt{V_{clos}}\ M\ \gamma : T \xrightarrow{\hat{\ell}_1, \hat{\ell}_2} S} \qquad \textit{Proxy}\ \frac{\mu \vdash v : S \xrightarrow{\hat{\ell}_1, \hat{\ell}_2} T}{\mu \vdash \mathtt{V_{proxy}}\ S \xrightarrow{\hat{\ell}_1, \hat{\ell}_2} T\ S' \xrightarrow{\hat{\ell}'_1, \hat{\ell}'_2} T'\ v : S' \xrightarrow{\hat{\ell}'_1, \hat{\ell}'_2} T'}$$

$$\textit{Ref}\ \frac{\mu[\langle n, \ell_1 \ell_2 \rangle] \equiv \langle T, v \rangle}{\mu \vdash \mathtt{V_{ref}}\ \langle n, \ell_1, \ell_2 \rangle : \mathtt{Ref}\ \ell_2\ T'} \qquad \textit{RefDyn}\ \frac{\mu[\langle n, \ell_1 \ell_2 \rangle] \equiv \langle T, v \rangle}{\mu \vdash \mathtt{V_{ref}}\ \langle n, \ell_1, \ell_2 \rangle : \mathtt{Ref}\ ¿\ T'}$$

$$\textit{Lab}\ \frac{\mu \vdash v : T \qquad \ell \preccurlyeq \ell'}{\mu \vdash \mathtt{V_{lab}}\ \ell\ v : \mathtt{Lab}\ \ell'\ T} \qquad \textit{LabDyn}\ \frac{\mu \vdash v : T}{\mu \vdash \mathtt{V_{lab}}\ \ell\ v : \mathtt{Lab}\ ¿\ T}$$

Fig. 7: Value typing

V. Comparing Gradual Security-Typed Language Designs

In this section we compare four noteworthy designs - λ_{gif} [24], *ML-GS* [25], GSL_{Ref} [26], and *GLIO* [1].

A. Language Features

Table I compares the features of the four languages, which fall into three categories: the design of the language (whether it provide implicit or explicit casts, which is the distinction between a gradually typed surface language versus a cast calculus that is meant to serve as an intermediate language), the heap model (how the language handles mutable references), and the labeling granularity (in what places information flow labels may appear).

a) Language design: λ_{gif} and *ML-GS* provide explicit casts but no implicit ones, so they are cast calculi. GSL_{Ref} provides implicit casts; it is a gradually-typed language that is derived from its statically-typed sister language SSL_{Ref}. *GLIO* also provides implicit casts as we discussed in section II. It does not have a statically-typed sister language.

b) Heap model: λ_{gif} does not have mutable reference. Both *ML-GS* and *GLIO* choose to store a value together with its type on heap and generate a cast from the heap type to the type of the reference at runtime. GSL_{Ref}, on the other hand, stores casts represented as *evidence* on the heap. Although the approach of *GLIO* sounds similar to the one of *ML-GS*, there is a type invariant of *GLIO* that *ML-GS* lacks. In *GLIO* the type of a cell stays the same across updates, which is enabled by reading the type from the address first, followed by casting the value into that type. As shown in figure 4, the T'' stays unchanged in the `set` case. The heap model of *ML-GS* does not enforce this invariant; the $R{-}Asgn$ rule makes it possible to completely replace the raw type of a cell, so the cast may fail when reading from a reference.

These heap models that insert casts at runtime when reading and writing create a challenge for assigning blame. *GLIO* does not perform blame tracking. On the other hand, *ML-GS* does perform blame tracking and propagates blame labels through the heap to assign blame when dereferencing. However, this creates a problem regarding the statement of the blame theorem, which usually says that if each cast labeled p in term M of the cast calculus is a safe cast, then M will not reduce to `blame` p. But the types stored on the heap are only known when a program executes. Although the authors of *ML-GS* conjecture that "an extension" of the blame theorem could hold, they do not provide a theorem statement or a proof.

c) Labeling granularity: *GLIO* has first-class labels since it is based on the *LIO* library, where labels are treated as values. *GLIO* also follows *LIO* and *HLIO* and employs coarse-grained labeling, in which not all values are labeled by default and a programmer need to use `to-label` to explicitly protect a value. However, it is proved that coarse-grained labeling is equally expressive as fine-grained labeling, where every value is labeled [30].

d) Language Feature Summary: Overall *GLIO* and GSL_{Ref} are the more feature-rich languages among the four. *GLIO* supports coarse-grained labeling and first-class labels, which makes it easier to migrate from legacy code that does not have information flow labels. However, *GLIO* does not perform blame tracking, so it can't satisfy a blame theorem. Adding blame tracking it challenging because of the heap model, though perhaps the ideas of Siek et al. [31] may be applicable. GSL_{Ref} offers insight into deriving the gradual security-typed language from its statically typed sister language. Although it lacks first-class labels, its fine-grained labeling scheme means a label comes with each value.

B. Theorems and Properties

Table II summarizes the metatheoretic properties that the four languages satisfy. The "maybe" option is for properties that are either conjectured by the authors or that we suspect they would satisfy. Noninterference and type safety are satisfied by all of the languages. We discuss the other three properties in further detail:

a) Gradual guarantees: The static gradual guarantee states lowering the type precision of a term does not introduce a static type error. The dynamic gradual guarantee states that lowering the type precision of a term does not change its runtime behavior. On the other hand, increasing the type precision of a term may trigger a cast error (e.g. by adding an incorrect type annotation) but otherwise the behavior remains unchanged. Neither λ_{gif} nor *ML-GS* discuss the gradual guarantees, as they preceeded the invention of the gradual guarantees. GSL_{Ref} satisfies the static gradual guarantee but

TABLE I: Comparison of language features

System	Implicit Casts	Explicit Casts	Mutable reference	First-class label	Labeling scheme
λ_{gif}	✗ No	✓ Yes	✗ No	✗ No	Fine
ML-GS	✗ No	✓ Yes	✓ Yes	✗ No	Fine
GSL_{Ref}	✓ Yes	✓ Yes	✓ Yes	✗ No	Fine
GLIO	✓ Yes	✗ No	✓ Yes	✓ Yes	Coarse

TABLE II: Comparison of language properties

System	Noninterference	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

not the dynamic gradual guarantee, which the authors claim is in tension with noninterference. *GLIO* resolves this tension by having casts check labels only, without *classifying* the data.

b) Blame theorem: The blame theorem says that if every cast labeled by p in a term M is safe by satisfying a subtyping relation, then M will not reduce to `blame` p. The authors of λ_{gif} proved the blame theorem but λ_{gif} lacks mutable references. The authors of *ML-GS* conjecture that the language satisfies a blame theorem but do not state the theorem or give a proof. Neither GSL_{Ref} nor *GLIO* performs blame tracking.

c) Space efficiency: The operational semantics of λ_{gif}, *ML-GS*, and *GLIO* both use function proxies that can build up, so they are not space efficient. GSL_{Ref} utilizes the AGT framework which can in principle enable space efficiency [32, 33], but space efficiency is not discussed in the GSL_{Ref} paper.

VI. Conclusion and Future Work

In this paper we briefly reviewed the design of a gradual security-typed language, *GLIO*, defined its semantics with a definitional interpreter and provided a mechanized proof of type safety in *Agda*. Based on our comparison and analysis of four existing language designs, we recommend that a gradual security-typed language have the following characteristics.

- **A gradual language and a cast calculus.** The language design should include both a gradual surface language with implicit casts and a cast calculus with explicit casts.
- **Information flow control with fine-grained labeling.** Since fine-grained labeling and coarse-grained labeling are equally expressive, we choose fine-grained labeling, where each value is labeled and (gradual) labels are embedded into the types, thus providing a more uniform syntax. Different from GSL_{Ref} but similar to *GLIO*, values should have a *concrete* label. An unlabeled value should be shorthand for a default low-security label. We conjuncture that this alleviates the problem with GSL_{Ref}, where values can become dynamically typed and lose their original label when changing type annotations to be less precise.
- **Embedding of static and dynamic information flow control.** The gradual language should include include both static and dynamic information flow control. This means that the NSU checks should be expressed as casts, similar to the path that GSL_{Ref} follows. We are able to formally define the static and dynamic extremes leveraging this approach.
- **Mutable reference and proxies.** The language should support mutable reference. However, unlike *ML-GS* and *GLIO*, the standard approach involving reference proxies and a simple heap (that maps addresses to values) should be investigated [34].
- **Blame tracking.** The language should support blame tracking.
- **Space efficiency.** The language should be space efficient.

We plan to investigate a language with these characteristics and develop its metatheory. Hopefully it will meet all the criteria discussed in section V-B.

References

[1] A. A. de Amorim, M. Fredrikson, and L. Jia, "Reconciling noninterference and gradual typing," in *Logic in Computer Science*, ser. LICS, July 2020.

[2] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland, "Refined criteria for gradual typing," in *SNAPL: Summit on Advances in Programming Languages*, ser. LIPIcs: Leibniz International Proceedings in Informatics, May 2015.

[3] W. Stallings, L. Brown, M. D. Bauer, and A. K. Bhattacharjee, *Computer security: principles and practice*. Pearson Education Upper Saddle River, NJ, USA, 2012.

[4] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*. IEEE, 1982, pp. 11–11.

[5] D. E. Bell and L. J. La Padula, "Secure computer system: Unified exposition and multics interpretation," MITRE CORP BEDFORD MA, Tech. Rep., 1976.

[6] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[7] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[8] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of computer security*, vol. 4, no. 2-3, pp. 167–187, 1996.

[9] S. A. Zdancewic and A. Myers, *Programming languages for information security*. Cornell University, 2002.

[10] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[11] G. Barthe and T. Rezk, "Non-interference for a jvm-like language," in *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, 2005, pp. 103–112.

[12] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A logic for information flow in object-oriented programs," *ACM SIGPLAN Notices*, vol. 41, no. 1, pp. 91–102, 2006.

[13] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive noninterference," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 79–90.

[14] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 228–241.

[15] F. Pottier and V. Simonet, "Information flow inference for ml," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002, pp. 319–330.

[16] V. Simonet and I. Rocquencourt, "Flow caml in a nutshell," in *Proceedings of the first APPSEM-II workshop*, 2003, pp. 152–165.

[17] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theoretical Computer Science*, vol. 411, no. 19, pp. 1974 – 1994, 2010, mathematical Foundations of Programming Semantics (MFPS 2006). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397510000502

[18] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in haskell," in *Proceedings of the 4th ACM symposium on Haskell*, 2011, pp. 95–106.

[19] ——, "Flexible dynamic information flow control in the presence of exceptions," *arXiv preprint arXiv:1207.1457*, 2012.

[20] P. Buiras, D. Vytiniotis, and A. Russo, "Hlio: Mixing static and dynamic typing for information-flow control in haskell," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 289–301. [Online]. Available: https://doi.org/10.1145/2784731.2784758

[21] J. G. Siek and W. Taha, "Gradual typing for functional languages," in *Scheme and Functional Programming Workshop*, September 2006, pp. 81–92.

[22] S. Tobin-Hochstadt and M. Felleisen, "Interlanguage migration: From scripts to programs," in *Dynamic Languages Symposium*, 2006.

[23] J. Matthews and R. B. Findler, "Operational semantics for multi-language programs," in *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2007.

[24] T. Disney and C. Flanagan, "Gradual information flow typing," in *International workshop on scripts to programs*, 2011.

[25] L. Fennell and P. Thiemann, "Gradual security typing with references," in *2013 IEEE 26th Computer Security Foundations Symposium*, June 2013, pp. 224–239.

[26] M. Toro, R. Garcia, and E. Tanter, "Type-driven gradual security with references," *ACM Trans. Program. Lang. Syst.*, vol. 40, no. 4, pp. 16:1–16:55, Dec. 2018. [Online]. Available: http://doi.acm.org/10.1145/3229061

[27] R. Garcia, A. M. Clark, and E. Tanter, "Abstracting gradual typing," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL 2016. New York, NY, USA: ACM, 2016, pp. 429–442. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837670

[28] A. Bove, P. Dybjer, and U. Norell, "A brief overview of agda — a functional language with dependent types," in *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, ser. TPHOLs '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 73–78.

[29] N. Amin and T. Rompf, "Type soundness proofs with definitional interpreters," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: ACM, 2017, pp. 666–679. [Online]. Available: http://doi.acm.org/10.1145/3009837.3009866

[30] V. Rajani and D. Garg, "Types for information flow control: Labeling granularity and semantic models," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 233–246.

[31] J. G. Siek, M. M. Vitousek, M. Cimini, S. Tobin-Hochstadt, and R. Garcia, "Monotonic references for efficient gradual typing," in *European Symposium on Programming*, ser. ESOP, April 2015.

[32] M. Toro and É. Tanter, "Abstracting gradual references," *Science of Computer Programming*, vol. 197, p. 102496, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642320301052

[33] F. Bañados Schwerter, A. M. Clark, K. A. Jafery, and R. Garcia, "Abstracting gradual typing moving forward: Precise and space-efficient," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3434342

[34] D. Herman, A. Tomb, and C. Flanagan, "Space-efficient gradual typing," in *Trends in Functional Prog. (TFP)*, April 2007, p. XXVIII.

APPENDIX A
DETAILED DEFINITIONS OF CASTS BETWEEN LABELS AND TYPES

Figure 8 shows the definitions of the cast functions. $\texttt{V}_{\texttt{clos}}\ M\ \gamma$ denotes a closure value with a well-typed body M and an environment γ. $\texttt{V}_{\texttt{proxy}}\ S \xrightarrow{\hat{\ell}_1,\hat{\ell}_2} T\ S' \xrightarrow{\hat{\ell}'_1,\hat{\ell}'_2} T'\ v$ stands for a function proxy - a value wrapped with two function types and a proof that the source is the consistent subtype of the target. `castL` guarantees the runtime program counter pc is the consistent subtype of the static one $\hat{\ell}_2$. `castT` casts a value v from T_1 to T_2. It checks whether T_1 is the consistent subtype of T_2. There are some additional label checks in the `Ref` $\hat{\ell}$ `T` and `Lab` $\hat{\ell}$ `T` cases. A reference value `V-ref` $\langle n, \ell_1, \ell_2\rangle$, whose secrecy is denoted by ℓ_2, can inhabit either `Ref ¿ T` or `Ref` ℓ_2 `T` for some T. Similarly, a labeled value `V-lab` ℓ `v` can inhabit `Lab` $\hat{\ell}$ `T` for some T provided that $\ell \precsim \hat{\ell}$. If any of these checks fails, the cast results in a `castError`. For example, if we attempt to make `V-lab High v` to inhabit `Lab Low T` for some v, T. When casting a value of function type, we use function proxy and wrap the source type, the target type, and a proof that the source is the consistent subtype of the target using `V-proxy`. In figure 3, in the function application case of $\mathcal{V}$, we have showed how a function proxy is consumed.

APPENDIX B
DETAILED LEMMAS AND PROOFS OF SECTION IV

A. Supplementary Lemmas

Lemma 3 (Variable lookup)**.** *If the environment is well-typed* $\Gamma; \mu \vdash \gamma$ *and* $\Gamma[x] \equiv T$*, then:*

$$\exists v. \gamma[x] \equiv v \wedge \mu \vdash v : T$$

Proof. By induction on the De Bruijn index x:

- If $x = 0$, the first value in a well-typed environment is always well-typed.
- If $x = suc\ k$ for some k, it is proved by the induction hypothesis about k.

■

Lemma 4 (Heap lookup)**.** *If the heap is well-typed* $\sigma \vdash \mu$ *and* $\mu[\langle n, \ell_1, \ell_2\rangle] \equiv \langle T, v\rangle$ *for some location* $\langle n, \ell_1, \ell_2\rangle$*, then* $\sigma \vdash v : T$.

Proof. By induction on $\sigma \vdash \mu$:

- If the heap is empty, this is impossible because there is no $\langle T, v\rangle$ pair.
- If the μ is not empty, there are two possibilities. If the location to lookup is the same as the index of the first cell, then v is typed at T since the heap is well-typed. If not, it is proved by the induction hypothesis about the rest of the heap.

■

Lemma 5 (Cast `castT'` is well-typed)**.** *If* $T_1 \lesssim T_2$*, with a well-typed heap* $\mu \vdash \mu$ *and a well-typed value* $\mu \vdash v : T_1$*, we have:*

$$\vdash castT'\ \mu\ pc\ T_1\ T_2\ \_\ v : T_2$$

Proof sketch. By induction on the typing derivation of the value v. Since v is well-typed, we never go into the `stuck` branches. The complete proof is mechanized in *Agda*. ■

Lemma 6 (Updating preserves well-typed heap)**.** *If heap* σ *is well-typed* $\mu \vdash \sigma$*, the address to update is valid* $\mu[\langle n, \ell_1, \ell_2\rangle] \equiv \langle T, v\rangle$*, the new value is well-typed* $\mu \vdash w : T$*, then:*

$$\langle n, \ell_1, \ell_2\rangle \mapsto \langle T, w\rangle :: \mu \vdash \sigma$$

Proof sketch. By induction on $\mu \vdash \sigma$. The complete proof is mechanized in *Agda*. ■

Lemma 7 (Creating new cell preserves well-typed heap)**.** *If heap* σ *is well-typed* $\mu \vdash \sigma$*, the index* n *is fresh, the new value is well-typed* $\mu \vdash v : T$*, then:*

$$\langle n, \ell_1, \ell_2\rangle \mapsto \langle T, v\rangle :: \mu \vdash \sigma$$

Proof sketch. By induction on $\mu \vdash \sigma$. The complete proof is mechanized in *Agda*. ■

B. Proving That the Interpreter Is Safe

The interesting cases in the proof of proposition 1 are detailed below.

Proof sketch. The complete proof is formalized in *Agda*. We only detail a few cases here. If the evaluation times out, it is trivially well-typed. Otherwise, by induction on the typing derivation of the term M:

- **Case variable (`‘ x`):** The environment γ is well-typed so this case is proved by lemma 3.
- **Case `if`:** By lemma 3, there are only two possibilities that x typed ‘$\mathbb{B}$ may correspond to: `V-true` or `V-false`, so the program does not get stuck during the lookup of x. If x is `V-true` and we go to the then-branch M, we case on the evaluation result of M - if M times out or errors by either `castError` or `NSUError` then the result is trivially well-typed; evaluating M does not get stuck because of the induction hypothesis. If it returns a machine configuration, we proceed with `castL`, which either errors by a `castError`, which is well-typed, or evaluates to a configuration. The final `castT` either errors by a `castError`, which is again trivially well-typed, or invokes `castT'`, the well-typedness of which is proved by lemma 5. The proof of the else-branch N follows the same structure.
- **Case `get`:** By lemma 3, the value that x with a reference type corresponds to must be of form `V-ref` $\langle n, \ell_1, \ell_2\rangle$, whose typing may follow either the *Ref* rule or the *RefDyn* rule (in figure 7). In either case, we conclude that $\mu[\langle n, \ell_1, \ell_2\rangle] \equiv \langle T, v\rangle$, so the interpreter does not get stuck due to a failed heap lookup. Since the result of `castT` is well-typed (a corollary of lemma 5), the

$$\boxed{\textbf{\textit{castL}} : (\mu : Store) \to (pc : \mathcal{L}) \to (\hat{\ell}_1, \hat{\ell}_2 : \hat{\mathcal{L}}) \to Result\ Conf}$$

$$castL\ \mu\ pc\ \hat{\ell}_1\ \hat{\ell}_2\ =\ \begin{cases} result\ \langle \mu, \mathtt{V_{tt}}, pc\rangle & \text{, if } pc \precsim \hat{\ell}_2 \\ error\ \mathtt{castError} & \text{, otherwise} \end{cases}$$

$$\boxed{\textbf{\textit{castT'}} : (\mu : Store) \to (pc : \mathcal{L}) \to (T_1, T_2 : \mathbb{T}) \to (v : Value) \to Result\ Conf}$$

$$\begin{aligned}
castT'\ \mu\ pc\ `\top\ `\top\ \mathtt{V_{tt}}\ &=\ result\ \langle \mu, \mathtt{V_{tt}}, pc\rangle \\
castT'\ \mu\ pc\ `\top\ `\top\ _\ &=\ error\ \mathtt{stuck} \\
castT'\ \mu\ pc\ `\mathbb{B}\ `\mathbb{B}\ \mathtt{V_{true}}\ &=\ result\ \langle \mu, \mathtt{V_{true}}, pc\rangle \\
castT'\ \mu\ pc\ `\mathbb{B}\ `\mathbb{B}\ \mathtt{V_{false}}\ &=\ result\ \langle \mu, \mathtt{V_{false}}, pc\rangle \\
castT'\ \mu\ pc\ `\mathbb{B}\ `\mathbb{B}\ _\ &=\ error\ \mathtt{stuck} \\
castT'\ \mu\ pc\ `\mathcal{L}\ `\mathcal{L}\ (\mathtt{V_{label}}\ \ell)\ &=\ result\ \langle \mu, \mathtt{V_{label}}\ \ell, pc\rangle \\
castT'\ \mu\ pc\ `\mathcal{L}\ `\mathcal{L}\ _\ &=\ error\ \mathtt{stuck}
\end{aligned}$$

$$castT'\ \mu\ pc\ (\mathtt{Ref}\ \hat{\ell}_1\ T_1')\ (\mathtt{Ref}\ \hat{\ell}_2\ T_2')\ (\mathtt{V_{ref}}\ \langle n, \ell_1, \ell_2\rangle)\ =\ \begin{cases} result\ \langle \mu, \mathtt{V_{ref}}\ \langle n, \ell_1, \ell_2\rangle, pc\rangle & \text{, if } \hat{\ell}_2 \text{ is ¿ or } \hat{\ell}_2 \equiv \ell_2 \\ error\ \mathtt{castError} & \text{, otherwise} \end{cases}$$

$$castT'\ \mu\ pc\ (\mathtt{Ref}\ \hat{\ell}_1\ T_1')\ (\mathtt{Ref}\ \hat{\ell}_2\ T_2')\ \_ =\ error\ \mathtt{stuck}$$

$$castT'\ \mu\ pc\ (\mathtt{Lab}\ \hat{\ell}_1\ T_1')\ (\mathtt{Lab}\ \hat{\ell}_2\ T_2')\ (\mathtt{V_{lab}}\ \ell\ v)\ =\ \begin{cases} do \\ \quad \langle \mu', v', pc'\rangle\ \leftarrow\ castT'\ \mu\ pc\ T_1'\ T_2'\ v \\ \quad result\ \langle \mu', \mathtt{V_{lab}}\ \ell\ v', pc'\rangle & \text{, if } \ell \precsim \hat{\ell}_2 \\ error\ \mathtt{castError} & \text{, otherwise} \end{cases}$$

$$castT'\ \mu\ pc\ (\mathtt{Lab}\ \hat{\ell}_1\ T_1')\ (\mathtt{Lab}\ \hat{\ell}_2\ T_2')\ \_\ =\ error\ \mathtt{stuck}$$

$$castT'\ \mu\ pc\ S\ \xrightarrow{\hat{\ell}_1, \hat{\ell}_2}\ T\ S'\ \xrightarrow{\hat{\ell}_1', \hat{\ell}_2'}\ T'\ v\ =\ \begin{cases} result\ \langle \mu, \mathtt{V_{proxy}}\ S\ \xrightarrow{\hat{\ell}_1, \hat{\ell}_2}\ T\ S'\ \xrightarrow{\hat{\ell}_1', \hat{\ell}_2'}\ T'\ v, pc\rangle \\ \qquad \text{, if } v \text{ is } \mathtt{V_{clos}} \text{ or } \mathtt{V_{proxy}} \\ error\ \mathtt{stuck} & \text{, otherwise} \end{cases}$$

$$\boxed{\textbf{\textit{castT}} : (\mu : Store) \to (pc : \mathcal{L}) \to (T_1, T_2 : \mathbb{T}) \to (v : Value) \to Result\ Conf}$$

$$castT\ \mu\ pc\ T_1\ T_2\ v\ =\ \begin{cases} castT'\ \mu\ pc\ T_1\ T_2\ v & \text{, if } T_1 \lesssim T_2 \\ error\ \mathtt{castError} & \text{, otherwise} \end{cases}$$

Fig. 8: Casts between labels and types

final result of `get` is well-typed (note that the heap is not modified by `castT` or `castL`).

Case `set`: The reasoning is similar to that of `get`. If the two casts both return valid configurations, we need to prove that the configuration after `setmem` is still well-typed. If the NSU check in `setmem` fails, an `NSUError` is well-typed. Otherwise, the well-typedness of the updated heap is justified by lemma 6.

Case `new-dyn`: Similar to `get` and `set`, we know that both variables correspond to well-typed values. If the NSU check fails, an `NSUError` is well-typed. Otherwise, the resulting configuration contains a heap extended with the new cell and a returned reference. The well-typedness of the extended heap is proved using 7. The reference is well-typed due to the *RefDyn* rule.

Case `new`: Similar to `new-dyn`.

■

APPENDIX C
COMPLETE DEFINITIONS AND PROOFS IN *Agda*

TABLE III: Mapping definitions and theorems in the paper to *Agda* code

Definition or theorem	**File**	***Agda* function or datatype**
Label join ($\sqcup$)	`StaticsGLIO.agda`	Operator `_⊔_`
Label meet ($\sqcap$)	`StaticsGLIO.agda`	Operator `_⊓_`
Label partial order ($\preccurlyeq$)	`StaticsGLIO.agda`	Operator `_≼_`
Gradual label join ($\curlyvee$)	`StaticsGLIO.agda`	Operator `_⋎_`
Gradual label meet ($\curlywedge$)	`StaticsGLIO.agda`	Operator `_⋏_`
Gradual label intersection ($\prod$)	`StaticsGLIO.agda`	Operator `_∏_`
Label consistent subtyping ($\precsim$)	`StaticsGLIO.agda`	Operator `_≾_`
Type join ($\vee$)	`StaticsGLIO.agda`	Operator `_∨_`
Type meet ($\wedge$)	`StaticsGLIO.agda`	Operator `_∧_`
Type intersection ($\cap$)	`StaticsGLIO.agda`	Operator `_∩_`
Consistent subtyping ($\lesssim$)	`StaticsGLIO.agda`	Operator `_≲_`
Typing rules	`StaticsGLIO.agda`	Datatype `_[_,_]⊢_⁈_`
Value and heap	`Store.agda`	Datatype `Location`, `Value`, `Cell`, and `Store`
Heap lookup	`Store.agda`	Function `lookup`
Machine configuration	`Interp.agda`	Datatype `Conf` and `Result`
Casts	`Interp.agda`	Function `castL` and `castT`
Interpreter (figure 8, 3, 4, and 5)	`Interp.agda`	Function $\mathcal{V}$
Value typing	`WellTypedness.agda`	Datatype `_⊢`$_v$`_⁈_`
Well-typed environment	`WellTypedness.agda`	Datatype `_|_⊢`$_e$`_`
Well-typed heap (store)	`WellTypedness.agda`	Datatype `_⊢`$_s$`_`
Well-typed result	`WellTypedness.agda`	Datatype `⊢`$_r$`_⁈_`
Variable lookup lemma 3	`WellTypedness.agda`	Function `⊢γ→∃v` and `⊢γ→⊢v`
Heap lookup lemma 4	`WellTypedness.agda`	Function `lookup-safe` and `lookup-safe-corollary`
`castT'` is well-typed (lemma 5)	`WellTypedness.agda`	Function `⊢castT′`
Heap update lemma 6	`WellTypedness.agda`	Function `ext-update-pres-⊢`$_s$
Heap new lemma 7	`WellTypedness.agda`	Function `ext-new-pres-⊢`$_s$
$\mathcal{V}$ is type safe (prop 1)	`InterpSafe.agda`	Function `𝒱-safe`
Type safe (theorem 2)	`TypeSafety.agda`	Function `type-safety`
Example in listing 1	`Example.agda`	Module `FunExample`
Examples in listing 2 and listing 4	`Example.agda`	Module `LabExample`
Examples in listing 5 and listing 6	`Example.agda`	Module `RefExample`