Reconciling noninterference and gradual typing

Arthur Azevedo de Amorim  
Carnegie Mellon University

Matt Fredrikson  
Carnegie Mellon University

Limin Jia  
Carnegie Mellon University

Abstract
One of the standard correctness criteria for gradual typing is the dynamic gradual guarantee, which ensures that loosening type annotations in a program does not affect its behavior in arbitrary ways. Though natural, prior work has pointed out that the guarantee does not hold of any gradual type system for information-flow control. Toro et al.’s GSL-Ref language, for example, had to abandon it to validate noninterference.

We show that we can solve this conflict by a feature of prior proposals: type-guided classification, or the use of type ascription to classify data. Gradual languages require run-time secrecy labels to enforce security dynamically; if type ascription merely checks these labels without modifying them (that is, without classifying data), it cannot violate the dynamic gradual guarantee. We demonstrate this idea with GLIO, a gradual type system based on the LIO library that enforces both the gradual guarantee and noninterference, featuring higher-order functions, general references, coarse-grained information-flow control, security subtyping and first-class labels. We give the language a domain-theoretic semantics, using Pitts’ framework of relational structures to prove noninterference and the dynamic gradual guarantee.

1 Introduction
Gradual type systems allow incomplete type annotations for combining the safety of static typing with the flexibility of dynamic languages. In the gradual λ-calculus of Siek and Taha [27], for example, we can declare the argument of a function \( f \) as an integer but omit its return type. This causes the type checker to reject an expression such as \( f(true) \) while accepting \( f(0) + 1 \), understanding that the latter will trigger a run-time error if \( f(0) \) returns a string. Many language features have been adapted to gradual typing, including references [29], polymorphism [2, 17, 21, 34], among others.

Unlike other approaches that mix static and dynamic typing, ascribing types in a gradual language should barely affect a program’s behavior, a property known as the dynamic gradual guarantee (DGG) [28]: the program might be rejected by the type checker or encounter more cast errors, but its output should not change from 0 to 1. Albeit natural, this isolation can be challenging for languages that strive for more than basic type safety. It had to be abandoned in a gradual variant of System F to enforce parametricity [34],1 and in the GSL-Ref language [33] to enforce noninterference.

1Recent work has managed to lift this restriction using ideas similar to ours [21]; cf. Section 7.

Sadly, the guarantee does not hold in any existing gradual language for information-flow control (IFC) [33].

The goal of this paper is to remedy the situation for IFC languages without giving up on noninterference. The difficulty, we argue, stems from what we call type-guided classification: the ability to classify values through static type annotations. This issue is illustrated in Figure 1, which shows a program in \( \lambda^{\text{info}} [4] \), a typical language for dynamic IFC. Values in \( \lambda^{\text{info}} \) carry a confidentiality label that is checked and propagated during execution to prevent information leaks. Unannotated values such as true are marked with a default label (in \( \lambda^{\text{info}} \), Public), which can be overridden with < S >. For example, the function \( f \) is given a Secret argument.

For now, ignore the commented type (* . . . *). If we ran this program in a typical language with no IFC checks, it would have the effect of leaking the secret input \( x \) through the reference \( z \), returning true when \( x = true \) and false when \( x = false \). Dynamic IFC prevents this breach with a discipline known as no-sensitive-upgrade (NSU) [4, 5, 31], which forbids updates to public references when the control flow is influenced by secrets. In \( f \), the reference \( y \) is implicitly labeled public because it is allocated in a public context and initialized with a public variable. This causes the NSU check to fail and terminate execution.

An extension of \( \lambda^{\text{info}} \) with gradual types could allow us to annotate \( b \) with the type in comments, declaring it as a secret boolean. What would this declaration mean? Current gradual IFC languages (GSL-Ref [33], ML-GS [12], etc.) interpret it as classification, thus setting \( b \)’s dynamic secrecy label to S. This causes the program to terminate successfully: \( b \)’s label is propagated to \( y \) and \( z \), the program accepts the two assignments (because \( x \) has the same secrecy as the references), and returns < S >true. Unfortunately, this behavior

Figure 1. Prototypical failure of the DGG due to NSU checks. The program throws an error when run, but successfully terminates if we uncomment the type annotation Bool<S>.

let f x =
  let b (* : Bool<S> *) = true in
  let y = ref b in
  let z = ref b in
  if x then y := false else ();
  if !y then z := false else ();
  !z

f (<S>true)
violates the DGG, because dynamic errors are not allowed to 
disappear when we provide type annotations.

This scenario suggests two possibilities for repairing the 
DGG: dropping the NSU discipline in favor of type-guided 
classification, or vice versa. The first option is problematic 
because it is hard to find other ways of enforcing noninter-
ference. One possibility would be to modify the semantics 
of conditionals so that they raise the secrecy of all refer-
ences that could be updated in either branch [25]. In Fig-
ure 1, this would mean raising \( y \)'s label above \( x \)'s even when 
the else branch is taken. Apart from the potential perfor-
mance impact, implementing this solution in any realistic 
language would require a rich analysis to compute write 
sets, which would likely push us further towards a static 
type system. And even if we decided that this was worth it, 
keeping type-guided classification would be problematic for 
another popular feature of IFC: first-class labels.

Labels are first class if they can be manipulated program-
matically; for instance, we might write \( \lambda b.10f b \Rightarrow \!) \) to 
test whether \( b \) holds a secret. First-class labels are often 
adopted in practically minded IFC systems [31, 36] because 
they enable rich data-dependent policies. Unfortunately, they 
can easily break the DGG with type-guided classification. 
Consider Figure 2, for instance: if the DGG were true, the 
unannotated program would behave the same way as the 
two annotated ones, which is impossible because they re-
turn different results. Similar issues have been observed in 
languages with dynamic type tests [9, 28]: if programs can 
test anything about a value's type, they can discern between 
different static annotations.

Thus, to reconcile noninterference and gradual typing, 
we are led to the second option: abandoning type-guided 
classification. The effect of an annotation should be merely 
to check labels, not to modify them. For Figure 1, this would 
mean that \( b, y \) and \( z \) would still be dynamically labeled \( P \) 
despite the static annotation \( \text{Bool}<S> \), triggering an NSU 
error without any harm to the gradual guarantee. Likewise, 
the annotations in Figure 2 could lead to a cast error, but 
they would not change the result of the test. Modifying labels 
should still be possible, but through a term-level operation 
that is not covered by the DGG.

We realize this idea with GLIO, a gradual language based 
on the LIO library [32]. LIO exposes an API for securely 
manipulating secret data, to which GLIO adds optional an-
notations for preventing security errors statically. Following 
the tradition of gradual typing, GLIO features a notion of con-
sistent subtyping to allow annotated and unannotated code 
to interoperate automatically, unlike prior work [10], where 
annotations might need to be checked manually. We still 
need to investigate if GLIO could be embedded in Haskell 
like LIO, but a standalone implementation should pose no 
challenges.

An important characteristic of gradual type systems is 
how much support they provide for transitioning legacy 
programs to richer type disciplines. The literature on gradual 
IFC offers different answers to this question; ML-GS [12], for 
example, requires references to be given an explicit secrecy 
label, and thus does not directly apply to legacy programs, 
while GSLRef [33] allows omitting all such annotations. By 
extending LIO, GLIO adopts a mixed stance in that regard. 
On the one hand, LIO does require programs to provide term-
level annotations for certain operations, including reference 
allocation. On the other hand, LIO's coarse-grained design 
obeys the need for tracking labels in most of the program; 
most values are protected by the PC label, a state component 
used in NSU checks.

In principle, it would be possible to allow missing label 
annotations for references in GLIO by choosing a default 
value for them, such as the current PC label. Unfortunately, 
the benefits of this approach would be limited for gradual 
typing: in the presence of first-class labels, no analog of the 
DGG can hold when overriding these defaults. We do not 
know if the situation fundamentally changes if first-class 
labels are absent, but missing reference annotations are not 
the only source of violations for the DGG: similar issues arise 
in GSLRef even if all reference annotations are present, by 
adapting the counterexample of Figure 1 to its syntax.

Our contributions, in sum, are as follows. We introduce 
GLIO, a gradual language based on LIO with higher-order 
functions and storage, flow-insensitive references, coarse-
gained IFC, security subtyping and public, first-class la-
bes. After an informal tour of the language in Section 2, 
we present its syntax and type system in Section 3, and 
define its semantics in Section 4. We prove that GLIO satis-
fies both termination- and error-insensitive noninterference 
(Section 5) and the gradual guarantee of Siek et al. [28] (Sec-
tion 6). We discuss related work in Section 7 and conclude 
in Section 8. Detailed proofs and definitions are included in 
the full version of this paper [6].

2 Overview

Before diving into technical details, we give a brief tour of 
GLIO. Traditionally, IFC languages have followed a fine-
grained discipline: every value carries a secrecy label, which 
is implicitly checked and propagated on every operation (stati-
cally or dynamically). This category includes \( \lambda^\text{info} \) [4]. Flow
Reconciling noninterference and gradual typing

Caml [23] and Jif [19], among others. By contrast, systems such as DCC [1], LIO [32] and GLIO follow a coarse-grained discipline: only certain values carry labels, and they must be manipulated using special primitives. The two styles are equally expressive [24, 35], but coarse-grained systems are easier to implement (since they track less information) and offer a smoother migration path to legacy programs (since most of the code does not need to worry about IFC).

Following LIO, GLIO places labeled values in a special type called Lab, and uses a monad LIO to express computations that handle secrets. Its most basic primitives are:

```haskell
label :: Label -> a -> LIO (Lab a)
unlabel :: Lab a -> LIO a
labLabel :: Lab a -> Label
pLabel :: LIO Label
```

The types shown here mimic those of the original LIO, but we’ll soon see that they can be refined with secrecy annotations. The `label` and `unlabel` functions are used to wrap a value of type `a` with a secrecy label and to unwrap it. To do this safely, the LIO monad encapsulates a state component known as the PC label, as usual in dynamic IFC. This label bounds the secrecy of all the values that have been unlabeled during the computation. Before assignments, the program performs an NSU check on this label to determine whether the operation is safe. The functions `labLabel` and `pLabel` allow inspecting the label of a labeled value and the current PC label.

The behavior of these primitives is illustrated in Figure 3, which shows a loose translation of Figure 1 into GLIO. In addition to the explicitly labeled values, the main difference with respect to Figure 1 is the new operator, which takes a secrecy label `P` as its argument. This translation is contrived for a coarse-grained system because of the spurious wrapping of the boolean `b`, but it is operationally closer to the original example and gives an idea of how GLIO enforces the DGG.

The program runs the same way as before. Unlabeling `b` amounts to a no-op: since its label is public, we do not need to update the PC label. On the contrary, `x` is marked as secret, so unlabeling it has the effect of bumping the PC label to `S`. This change is detected by GLIO’s NSU check, which deems the update to `y` unsafe and halts the program with an error.

Instead of `Lab Bool`, we could have given `b` the more precise type `Lab[S] Bool`, which says that the dynamic secrecy of the wrapped boolean is bounded by `S`. Since this label is `P`, which is below `S`, the assignment can be performed safely. Importantly, this does not modify `b`’s label, and updating `y` leads to the same result as before: an error. Since the behavior of the program did not change after refining the type, the DGG has not been violated.

The annotation did not break the DGG, but it was also not strong enough to catch the IFC error statically. Figure 4 demonstrates how this could be done in GLIO with a fully annotated version of the previous program. As in HLIO [10], the annotations on the LIO monad provide upper bounds on the PC label at the beginning and at the end of the computation. The annotations on `Ref` are stricter than those for `Lab`: instead of an upper bound, they give the exact secrecy of the contents the reference. This is to ensure safety: if the static label of a reference, `S`, were above its actual dynamic label, say `P`, the NSU check would still throw an error at run time, which the type checker would not be able to prevent.

To check `unlabel`, the type system propagates the static label of its argument into the PC label. Since `x` could be a secret, the type system rejects the assignment to `y`, as it could lead to an illegal implicit flow.

Figure 5 presents a middle ground between dynamic and static enforcement, using label introspection to test whether the NSU check would fail. Unlike labeled values, dynamic labels are themselves public, and can be inspected without tainting the PC. The `lab` operator computes the join, or least upper bound, of two labels, while `canFlowTo` checks if one label is below another. If the test passes, the assignment is performed without triggering any errors. Otherwise, the program logs the unsafe condition so that more robust recovery code can act later.

**Labeling and allocation.** Figure 6 further details the role of labels in values and references. The first program, `refLab`, stores the contents of a labeled value `x` in a fresh reference `r`. In this example, the new reference is typed as `Ref[S] Bool` because the annotation is constant, but in general this argument can be an arbitrarily complex expression, in which case the reference would get the imprecise type `Ref Bool`. For the allocation to succeed, the reference label must be above the PC label, which can be statically enforced in this case thanks to the PC annotations.
Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia

h x = do
  -- PC label = P
  b :: Lab[P] Bool <- label P True
  b' :: Bool <- unlabel b
  y :: Ref[P] Bool <- new P b'
  z :: Ref[P] Bool <- new P b'
  x' :: Bool <- unlabel x
  -- PC label = S
  if x'
    -- Assignment is rejected
    then set y False
    else return ()
  y <- get y
  if y then set z False
  else return ()
  get z

  do { x <- label S True; g x }

Figure 4. A fully annotated version of Figure 3 that is rejected at compile time

maybeUpdate :: Ref Bool -> Lab Bool -> LIO ()
maybeUpdate r x = do
  lpc <- pcLabel
  let lx = labLabel x
  let lr = refLabel r
  if lpc `lub` lx `canFlowTo` lr then do
    x' <- unlabel x
    set r x'
  else set errorOccurred True

Figure 5. Error prevention through label introspection

refLab x = do
  r :: Ref[S] Bool <- new S true
  -- toLab :: Label -> LIO a -> LIO (Lab a)
tolab S (do { x' <- unlabel x; set r x' })
return r

labRef :: Ref Bool -> LIO[P,P] (Lab Bool)
labRef r = toLab (refLabel r) (get r)

eqRef :: Ref Bool -> Ref Bool -> LIO[P,P] Bool
eqRef r1 r2 = return (r1 == r2)

Figure 6. Labeling and dynamic allocation

The function uses another primitive of GLIO, toLab, to avoid raising the PC label too much and causing spurious NSU errors—a problem known in the literature as label creep.

labCast :: LIO (Lab[P] Bool)
labCast = do
  b :: Lab[P] Bool <- label P True
  return (b :: Lab[S] Bool :: Lab Bool :: Lab[P] Bool)

labClass :: LIO (Lab[P] Bool)
labClass = do
  b :: Lab[P] Bool <- label P True
  b' <- unlabel b
  b'' <- label S b'
  return (b' :: Lab Bool :: Lab[P] Bool)

refCast :: LIO (Ref[S] Bool)
refCast = do
  r :: Ref[P] Bool <- new P True
  return (r :: Ref Bool :: Ref[S] Bool)

Figure 7. Casts in GLIO

The first argument of toLab is a label l that bounds the confidentiality of the result, and its second argument is a computation f. If the final PC label after running f is below l, toLab wraps the result in a value labeled with l and restores the PC label to its original value; otherwise, it throws an error. In refLab, the annotations are enough to guarantee the absence of errors and indicate that the PC label is indeed restored at the end of execution.

The second program, labRef, goes in the opposite direction: it uses toLab to wrap the contents of r into a labeled value of the same secrecy as r.

Fine-grained IFC often makes a distinction between the label of a reference, which protects its identity, and the label of its contents. In GLIO, what is sometimes called the "label of the reference" refers actually to the label of its contents: the identity of the reference is always public with respect to the PC label, and does not need to be protected with special checks. This is illustrated in the third program, eqRef, which tests if two references are identical. This comparison does not take their contents into account, which is why the PC label does not have to be tainted.

Casts and classification. GLIO includes a notion of consistent subtyping to allow annotated and unannotated code to interoperate. For example, we may pass a value r of type Lab Bool to refLab in Figure 6, and the language inserts the appropriate dynamic checks to ensure safety. (In this case, the checks are guaranteed to succeed, assuming the argument’s static label S denotes maximum secrecy.)

You may wonder why the first argument of toLab is needed, since we could have also used the final PC label to wrap the result. The problem is that labels in GLIO are public, and can be used to leak secrets [16]. By fixing the final label from the onset, we avoid the issue.
We can also trigger casts explicitly using type ascription, as shown in Figure 7. The first function, LabCast, labels the boolean True with $P$ and sends it through a series of casts, indicated with the :: operator. The type system checks each cast to rule out obvious or potential errors, such as coercing Bool to Unit or Lab[$S$] Bool to Lab[$P$] Bool.

Once LabCast reaches the last cast to Lab[$P$] Bool, it successfully returns True labeled as $P$, because the final label on the boolean stays the same across the casts—in other words, classification and type casts are decoupled. This contrasts with previous work [12, 13], in particular with GSLRef [33], which by design would trigger a run-time error, since it treats with another call to label which by design would trigger a run-time error, since it treats

with previous work [12, 13], in particular with

would be a lattice of labels

usual constructs for manipulating booleans, functions, and

let a sequence of

compact semantics later, we present the syntax in A-normal

a single monad (cf. Section 4). Second, to allow for a more

plification allows us to model pure and impure code with

guage still needs to be managed monadically, and this sim-

seen earlier. Because of cast errors, “pure” code in our lan-

As usual in gradual languages, the missing annotations in concrete syntax formally correspond to the gradual label $\in L \ equiv L \cup \{?\}$, which represents a statically unknown label.

The language does not include product, sum, and recursive types, but we foresee no difficulties in doing so—for recursive types in particular, GLIO already includes a higher-order store, which forces us to handle similar technical challenges.

Figure 9 presents the type system. The label indices in judgments $\Gamma \vdash_{l_1,l_2} e : T$ correspond to the static annotations on the LIO monad of Section 2: they constrain the PC label at the beginning and at the end of the execution of $e$. The rules reflect the behavior of the programs described earlier. For example, the variable rule does not change the label annotation because variables are already protected by the current PC label, and thus require no additional tainting. A similar reasoning applies to the introspection primitives refLabel, labLabel and pcLabel.

The rule for let shows how the label indices are threaded through as the computation unfolds. The consistent subtyping assumption $T' \prec T$ allows weakening security annotations or even omitting them entirely. Its definition, shown in Figure 10, resembles the subtyping discipline of Rajani and Garg [24], but adapted to the gradual setting using the Abstracting Gradual Typing (AGT) framework [15]. In AGT, a gradual type $T$ is interpreted as a set $\gamma(T)$ of fully annotated types, where each missing annotation is replaced by all possible completions. For example, $\gamma(\text{Lab}(\text{Bool}))$ is $\{\text{Lab}(\text{Bool}) | l \in L\}$. (The full version contains the complete definition [6].) This allows us to lift arbitrary predicates on fully annotated types to gradual types: the inductive presentation of Figure 10 is equivalent to saying that $T \prec S$ holds precisely when there exist $T' \in \gamma(T)$ and $S' \in \gamma(S)$ such that $T' \prec S'$, for a suitable subtyping relation $\prec$ on fully annotated types. The $\prec$ relation on $L$, which extends the one on $L$, can be recast in the same way, by posing $\gamma(?) = L$ and $\gamma(l) = \{l\}$.

On multiple rules, the consistent ordering on gradual labels is used to rule out IFC errors. For example, the side condition on the set rule subsumes the corresponding NSU check. Other rules, such as get and if, taint types and the PC label using partial consistent join operations $\&$ (Figure 11). The definition uses a consistent meet operation $\&$ and an intersection operation $\cap$ on types and gradual labels. These operations are not joins and meets in the usual sense, since the consistent orders are not transitive, and thus not actual orders; nevertheless, we can show

\[
\bar{l}_1 \& \bar{l}_2 \prec \bar{l}_1 \cap \bar{l}_2 \quad \text{and} \quad T_1 \& T_2 \prec T_1 \cap T_2
\]
\[ l \in L \]
\[ \bar{l} \in L \Leftrightarrow \{?\} \]
\[ b \in \{0, 1\} \]
\[ c \in L \cup \{x, y, z, \ldots\} \]
\[ \oplus \in \{\times, \varphi\} \]
\[ \Gamma \in \text{Var} \rightarrow \text{Type} \]
\[ e : T \Leftrightarrow \text{let}(e_1, x : T.e_2) \mid \text{unit} \mid b \mid \text{if}(x, e_2, e_0) \mid \text{fun}(x : T.e) \quad \text{standard} \]
\[ \mid \text{app}(x, y) \mid \text{get}(x) \mid \text{set}(x, y) \mid \text{new}(c, y) \mid \text{eqRef}(x, y) \]
\[ \mid \text{refLabel}(x) \mid \text{labLabel}(x) \mid \text{pcLabel}(c) \quad \text{IFC specific} \]
\[ | l | \oplus y \mid x \leq y \mid \text{unlabel}(x) \mid \text{toLab}(c, e) \]

**Figure 8. Syntax of terms and types**

\[
\begin{align*}
\Gamma(x) &= T & \Gamma \vdash_{\bar{l}, i} e_1 : T' & \Gamma[x \mapsto T] \vdash_{\bar{l}, i} e_2 : S & T' \ll T \\
\Gamma(x) &= \text{Bool} & \Gamma \vdash_{\bar{l}, i} e_1 : T_1 & \Gamma \vdash_{\bar{l}, i} e_0 : T_0 & \Gamma \vdash_{\bar{l}, i} \text{set}(x, y) : \text{Unit} & \Gamma \vdash_{\bar{l}, i} \text{get}(x) : T \\
\Gamma(x) &= \text{Ref}_i(T_1) & \Gamma(y) &= T_2 & T_2 \ll T_1 & \bar{l}_1 \ll \bar{l} & \Gamma \vdash_{\bar{l}, i} \text{new}(l_1, y) : \text{Ref}_i(T) & \Gamma \vdash_{\bar{l}, i} \text{Label} & \Gamma(y) &= T \\
\Gamma \vdash_{\bar{l}, i} \text{eqRef}(y, x) : \text{Bool} & \Gamma \vdash_{\bar{l}, i} \text{app}(f, x) : S & \Gamma \vdash_{\bar{l}, i} \text{refLabel}(x) : \text{Label} & \Gamma \vdash_{\bar{l}, i} \text{labLabel}(x) : \text{Label} & \Gamma \vdash_{\bar{l}, i} \text{Label} & \Gamma \vdash_{\bar{l}, i} e : S \\
\Gamma \vdash_{\bar{l}, i} \text{Label} & \Gamma \vdash_{\bar{l}, i} l : \text{Label} & \Gamma \vdash_{\bar{l}, i} \text{Label} & \Gamma \vdash_{\bar{l}, i} x : \text{Label} & \Gamma \vdash_{\bar{l}, i} \text{Label} & \Gamma \vdash_{\bar{l}, i} e : T \\
\Gamma \vdash_{\bar{l}, i} \text{toLab}(l_1, e) : \text{Label}(T) & \Gamma \vdash_{\bar{l}, i} \text{toLab}(l_2, e) : \text{Label}(T)
\end{align*}
\]

**Figure 9. Typing rules**

for \( i \in \{1, 2\} \), whenever the result of these operations is defined. Note that when all labels are \( ? \), \( T \ll S \) is equivalent to \( T = S \), so consistent joins become trivial and the type system reduces to a simplified version of LIO.

The two variants of new and toLab use different typing rules because the secrecy of their results is determined by their label argument. When this label is statically known (that is, in the new\((l, -)\) and toLab\((l, -)\) variants), the type system uses it in the result type. When this label is chosen dynamically, the result type is labeled with \( ? \).

The rule for toLab is slightly more permissive than the corresponding dynamic checks in LIO [32], which would translate as \( l_3 \ll l_1 \) instead of \( \bar{l}_3 \ll l_1 \lor \bar{l}_2 \). Intuitively, our variant is sound because the result of toLab is protected by both the ascribed label \( l_1 \) and the initial PC label \( \bar{l}_2 \). In Section 4, we will see that toLab takes the PC label into account during execution too.

### 4 Semantics

Each type \( T \) in GLIO corresponds to a set \( \llbracket T \rrbracket \) (Figure 12). As the heap can store arbitrary values, \( \llbracket T \rrbracket \) contains negative recursive occurrences, which requires some care to handle. To solve this issue, we define \( \llbracket T \rrbracket \) as a CPO rather than a plain set, by solving a domain equation [30]. We briefly
Reconciling noninterference and gradual typing

\[ \text{Ref}_{i}(T_{1}) \preceq \text{Ref}_{j}(T_{2}) \]

\[ \text{Lab}_{i}(T_{1}) \preceq \text{Lab}_{j}(T_{2}) \]

\[ \text{LIO}_{i, j}(X) \equiv \{ f : \text{Mem} \times L \times Y \to \text{Error} \} \]

\[ \text{Type}^{o} \equiv \{ T \in \text{Type} | T^{o} = T \} \]

\[ \text{valid}(l_{1}, m_{1}, m_{2}) \equiv \forall (T, r) \in \text{dom}(m_{2}). l_{1} \preceq r_{\text{label}} \land (l_{1} \neq r_{\text{stamp}} \Rightarrow (T, r) \in \text{dom}(m_{1})) \]

\[ \text{LIO}_{i, j}(X) \equiv \{ f : \text{Mem} \times L \times Y \to \text{Error} \} \]

\[ \text{valid}(l_{1}, m_{1}, m_{2}) \equiv \forall (T, r) \in \text{dom}(m_{2}). l_{1} \preceq r_{\text{label}} \land (l_{1} \neq r_{\text{stamp}} \Rightarrow (T, r) \in \text{dom}(m_{1})) \]

\[ \text{Type}^{o} \equiv \{ T \in \text{Type} | T^{o} = T \} \]

\[ \text{valid}(l_{1}, m_{1}, m_{2}) \equiv \forall (T, r) \in \text{dom}(m_{2}). l_{1} \preceq r_{\text{label}} \land (l_{1} \neq r_{\text{stamp}} \Rightarrow (T, r) \in \text{dom}(m_{1})) \]

\[ \text{Ref}_{i}(T_{1}) \preceq \text{Ref}_{j}(T_{2}) \]

\[ \text{Lab}_{i}(T_{1}) \preceq \text{Lab}_{j}(T_{2}) \]

\[ \text{LIO}_{i, j}(X) \equiv \{ f : \text{Mem} \times L \times Y \to \text{Error} \} \]

\[ \text{valid}(l_{1}, m_{1}, m_{2}) \equiv \forall (T, r) \in \text{dom}(m_{2}). l_{1} \preceq r_{\text{label}} \land (l_{1} \neq r_{\text{stamp}} \Rightarrow (T, r) \in \text{dom}(m_{1})) \]

\[ \text{Ref}_{i}(T_{1}) \preceq \text{Ref}_{j}(T_{2}) \]

\[ \text{Lab}_{i}(T_{1}) \preceq \text{Lab}_{j}(T_{2}) \]

\[ \text{LIO}_{i, j}(X) \equiv \{ f : \text{Mem} \times L \times Y \to \text{Error} \} \]

\[ \text{valid}(l_{1}, m_{1}, m_{2}) \equiv \forall (T, r) \in \text{dom}(m_{2}). l_{1} \preceq r_{\text{label}} \land (l_{1} \neq r_{\text{stamp}} \Rightarrow (T, r) \in \text{dom}(m_{1})) \]

\[ \text{Ref}_{i}(T_{1}) \preceq \text{Ref}_{j}(T_{2}) \]

\[ \text{Lab}_{i}(T_{1}) \preceq \text{Lab}_{j}(T_{2}) \]

\[ \text{LIO}_{i, j}(X) \equiv \{ f : \text{Mem} \times L \times Y \to \text{Error} \} \]

\[ \text{valid}(l_{1}, m_{1}, m_{2}) \equiv \forall (T, r) \in \text{dom}(m_{2}). l_{1} \preceq r_{\text{label}} \land (l_{1} \neq r_{\text{stamp}} \Rightarrow (T, r) \in \text{dom}(m_{1})) \]

---

**Figure 10.** Consistent subtyping

\[ \text{Ref}_{i}(T_{1}) \preceq \text{Ref}_{j}(T_{2}) \]

\[ \text{Lab}_{i}(T_{1}) \preceq \text{Lab}_{j}(T_{2}) \]

**Figure 11.** Gradual meets, gradual joins and intersections for labels and types. Most combinations of types yield undefined results. Here, \( \oplus \) stands for either \( \vee \) or \( \land \), and \( \otimes \) stands for the other operation.

---

review basic notions needed to cover the main contributions; interested readers can refer to the full version \[6\] for details.

First, by CPO we mean a partially ordered set where all increasing chains \( x_{0} \leq x_{1} \leq \cdots \) have a least upper bound \( \bigcup_{i \in \mathbb{N}} x_{i} \). The notation \( X \rightarrow Y \) refers to the CPO of continuous functions between \( X \) and \( Y \)—that is, monotone functions \( f : X \rightarrow Y \) such that \( f(\bigcup_{i \in \mathbb{N}} x_{i}) = \bigcup_{i} f(x_{i}) \), ordered pointwise. The lifted CPO \( X_{\bot} \) extends the CPO \( X \) with a least element \( \bot \), which represents nontermination. We use equality, or the discrete order, on CPOs such as \( \text{Ref}_{i} \), Type, \( L \) and its subsets. Error(\( X \)) is ordered pointwise. The order \( m_{1} \leq m_{2} \) on Mem holds when \( \text{dom}(m_{1}) = \text{dom}(m_{2}) \) and \( \forall T, r. m_{1}(T, r) \leq m_{2}(T, r) \).

Let us explain these definitions before moving on to the semantics of terms. The CPOs \( \text{Lab}(T) \) contain elements of \( X \) protected by a dynamic label \( l \); as explained in Section 2, this label is bounded by the annotation \( l \), not necessarily equally to it. A reference \( r = (r_{n}, r_{\text{stamp}}, r_{\text{label}}) \) carries two labels: \( r_{\text{stamp}} \) corresponds to the PC label at the moment of allocation, and \( r_{\text{label}} \) corresponds to the secrecy of its contents. As noted in Section 2, \( r_{\text{label}} \) must exactly match the static annotation.
The memory updates take as inputs a memory (\text{LIO}) and a PC label (\text{HLIO}). Its elements are functions that include it explicitly in the denotation of each type. Returning updates instead of the final memory is unimportant for program behavior, but it simplifies the proof that maps a type and a reference to a value \(v \in \Gamma\). We assume that \(T\) has no label annotations, because our semantics doesn’t track this information for stored values (we discuss a more efficient approach below). The predicate valid(\text{l}_i, m_1, m_2) describes which memory updates are allowed under the PC label \text{l}_i: new and updated locations must pass the NSU check for \text{l}_i (\text{l}_i \triangleleft \text{r}_\text{label})", and stamps must reflect their allocation context, which, as hinted earlier, is a technical device to simplify the noninterference proof.

The definition of LI\O does not preclude computations that access undefined locations in memory, because its elements take all possible memories as their input. It would be possible to rule out these errors with a Kripke semantics in the style of Levy [18], but the issue is orthogonal to our purposes, and we stick to the current formulation for simplicity. Note, however,
that some memory-related errors are ruled out by the shape of the memory. For instance, if we try to read \( m(\text{Bool}, r) \) and that location is defined, we know that it contains indeed a boolean, which we can access directly.

With the interpretation of types at hand, we are ready for the semantics of typed terms, shown in Figures 13 and 14. We equip LIO with the structure of a parameterized monad [3] (Figure 15), which we use to interpret the Haskell-like do notation in the definitions. Notice how bind applies updates to the initial memory before invoking its continuation, in accordance with our treatment of state. Figure 16 defines the interpretation of subtyping coercions. As explained earlier, coercing a value into Lab or Ref types never changes its label, only checks it, which will be important for the gradual guarantee. Similarly, the coercions triggered by casting or applying a function never modify the PC label.

**Figure 14.** Semantics of typing derivations (continued)

**Figure 15.** Monadic operations of LIO
Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia

\[ T \Rightarrow S \]_L : [T] \overset{\text{cont}}{\rightarrow} \text{LIO}_{\text{L}}(S) \quad (\text{for } T \Rightarrow S)

\[ T \Rightarrow T \]_L = \text{return} \quad (\text{for } T \in \{\text{Unit, Bool, Label}\})

\[ \text{Ref}_{L_1}(T) \Rightarrow \text{Ref}_{L_2}(S)(n, l_1, l_2) \overset{\text{def}}{=} \begin{cases} \text{return}(n, l_1, l_2) & \text{if } l_2 \in \gamma(l_2) \\ l(\cdot), \text{error} & \text{otherwise} \end{cases} \]

\[ \text{Lab}_{L_1}(T_1) \Rightarrow \text{Lab}_{L_2}(T_2)(v@l) \overset{\text{def}}{=} \begin{cases} \text{do} \{ v' \leftarrow \text{Ref}_{L_1}(T_1)(v); \text{return}(v'@l) \} & \text{if } l \in \downarrow l_2 \\ l(\cdot), \text{error} & \text{otherwise} \end{cases} \]

\[ T \overset{\bar{L}_1 \bar{L}_2}{\rightarrow} S \Rightarrow T' \overset{\bar{L}_1 \bar{L}_2}{\rightarrow} S'(f) \overset{\text{def}}{=} \begin{cases} \text{return } \lambda x'. \text{do} \{ x \leftarrow T'(x'); \text{Lab}_{L_1}(T_1)(\downarrow l_1); y \leftarrow f(x); \text{Lab}_{L_2}(T_2)(\downarrow l_2); S \leftarrow S'(y) \} \end{cases} \]

\[ \bar{L}_1 \Rightarrow \bar{L}_2(L) \overset{\text{def}}{=} \begin{cases} (\emptyset, 1, \bar{L}_1) & \text{if } L_1 \in \downarrow l_2 \\ \text{error} & \text{otherwise} \end{cases} \]

**Figure 16.** Label and type coercion

The behavior of basic ML operations is standard, except for coercions and the NSU checks in set and new. To read a reference, we cast its contents to ensure that the labels on the type are respected; conversely, when updating it reference, we use a cast to forget the labels. (Note that \( T \Rightarrow T' \) and \( T' \Rightarrow T \) hold for every \( T \).) A more efficient approach would be to use monotonic references [29], whose types are guaranteed to be bounded in precision by the type of their contents during execution. This property ensures that accesses to a reference of fully annotated type can be performed directly, without any casts. We believe that monotonic references could be incorporated in GLIO without compromising our results, but arguing about their correctness requires an intricate stateful invariant, and we keep our scheme for simplicity. Note that in the case of base types, the casts reduce to the identity, because they have no labels to be checked.

The IFC operations are modeled after their analogues in LIO [31], but toLab includes the initial PC label \( l_2 \) in its side condition, as anticipated by its typing rule. Note how unlabel and get taint the PC label to track the secrecy of the result.

The examples of Section 2 have already exercised the most interesting aspects of the semantics, except for one: stamps. Consider the following program \( e \), written in informal syntax for clarity (recall that \( S \) stands for \( T \)).

```
  toLab S $ do
      b' \leftarrow \text{unlabel } b
      \text{if } b' \text{ then do } \{ \text{new } S \text{ True; return } () \} 
      \text{else return } () \}
  \text{new } S \text{ True}
```

We can produce a typing judgment \([b \mapsto \text{Lab}_{\cdot}(\text{Bool})] \Rightarrow_{\perp, \perp} e : \text{Ref}_{\cdot}(\text{Bool})\)\], which corresponds to a function \([e] \) of type \(\text{Lab}_{\cdot}(2) \overset{\text{cont}}{\rightarrow} \text{LIO}_{\perp, \perp}(\text{Ref}_{\cdot})\). By running this program on two different inputs and an empty memory, we obtain successful executions

\[
\begin{align*}
[e](1@\top)(\emptyset, \perp) &= ((r_0 \mapsto 1, r_1 \mapsto 1), r_1, \perp) \\
[e](0@\top)(\emptyset, \perp) &= ((r_1 \mapsto 1), r_1, \perp),
\end{align*}
\]

where \(r_0 = (\text{Bool},(0, \top, \top))\) is allocated inside the conditional, and \(r_1 = (\text{Bool},(0, \perp, \top))\) is allocated at the end.

Although the secret \( b \) caused \( e \) to perform different allocations, the result is the same: the stamps allow us to perform the allocations in high-secrecy contexts without impacting references allocated in low-secrecy contexts. This technique, due to Azevedo de Amorim et al. [5], simplifies the proof of noninterference because we can match references in related executions up to equality. Without stamps, noninterference would still hold, but the values returned in each execution would not necessarily be equal, requiring a more complex argument to relate syntactically different references [8].

\section{5 Noninterference}

With the semantics pinned down, we are ready for our first main result: showing that GLIO satisfies termination- and error-insensitive noninterference. Informally, an attacker cannot tell the difference between two successful runs of a program that differ only on their secret inputs. To formalize this claim, we follow Abadi et al.’s work on DCC [1] and define a family of relations \((R_t)_{t \in \mathcal{L}}\) that characterize what elements of \([T]\) are indistinguishable to an observer bounded...
by \( I \) (Figure 17). The definition is again circular, but it can be solved with Pitts’ framework of relational structures [7, 22], as explained in the full version [6].

For base types and references, being indistinguishable simply means being equal. There are two notions of indistinguishability for \( \text{Lab}_I(X) \): weak (\( \approx_I \)) and strong (\( \equiv_I \)). Weak indistinguishability is only an auxiliary notion used to define indistinguishability for computations (\( \text{LIOM}_{i, i}(X) \)). We use two notions because GLIO guarantees that the label of a labeled value reveals nothing about the value, whereas the PC label at the end of a computation might reveal something about its result. An observer bounded by \( I \) can distinguish two memories if they differ either in their sets of low-stamp locations, \( \text{dom}_I \), or in two values stored at a low location.

Our goal is to prove \( [e] \approx_I [e] \) for every well-typed program \( e \). This implies that programs do not leak secrets; for example, if \( I = \bot \) and \( e : \text{Lab}_I(\text{Bool}) \rightarrow \text{Bool} \), we find that \( [e]([1]@T)(\emptyset, L) \) and \( [e]([0]@T)(\emptyset, \bot) \) output the same boolean if both terminate successfully.

**Theorem 5.1 (Noninterference).** If \( \Gamma \vdash_{i, i} e : T \), we have

\[
[e] \approx_I [e] : \Gamma \quad \Rightarrow \quad \text{LIOM}_{i, i}(\Gamma[[T]]).
\]

**Sketch.** By induction on the typing derivation of \( e \). The semantics of the language is defined by using the monadic interface of Figure 15 to compose the operations in Figures 14 and 16. Thus, we just have to show that indistinguishability is preserved by these operations and under composition. \( \square \)

### 6 Gradual guarantees

The main novelty of GLIO is that it satisfies the *dynamic gradual guarantee* [28]: making label annotations more precise

By induction on the typing derivation of \( e \). The semantics of the language is defined by using the monadic interface of Figure 15 to compose the operations in Figures 14 and 16. Thus, we just have to show that indistinguishability is preserved by these operations and under composition. \( \square \)

![Figure 18. Syntactic dynamism relations](image-url)
Theorem 6.1 (Dynamic Gradual Guarantee, Simple). Suppose that $e < e'$ with $\prec_\perp \perp e : T$ and $\prec_\perp \perp e' : T'$.

- If $[e](0)(\perp, \perp) = \perp$, then $[e'](0)(\perp, \perp) = \perp$.
- If $[e](0)(\perp, \perp) = (m, v, l)$, then there exist $m'$ and $v'$ such that $[e'](0)(\perp, \perp) = (m', v', l)$.

The premise $e < e'$, defined on Figure 18, says that $e'$ is obtained from $e$ by replacing some labels on type annotations with $\perp$. The conclusion says that $e$ and $e'$ must behave similarly, except when $e$ throws an error, in which case $e'$ can do whatever it wants. In particular, $e'$ can only fail if $e$ does.

GLIO also satisfies the static gradual guarantee, which says that removing label annotations from a term does not break type checking.

Theorem 6.2 (Static Gradual Guarantee). If $\Gamma \triangleright l, \perp T$, $e < e'$, and $\Gamma \vdash e : T$, there exist $l_2 \triangleright l_1$ and $T' \triangleright T$ such that $\Gamma' \vdash l_2 : e' : T'$.

The proof of this result is a straightforward induction on the typing derivation. Theorem 6.1, on the other hand, requires more care, as the statement is not strong enough to be established directly by induction. We use a generalization similar to prior formulations of the DGG [20, 21].

The error approximation relations $[e] < [e']$ in the conclusion are defined on Figure 19. Like indistinguishability in Section 5, they are constructed using Pits’ work [7, 22]. A technical subtlety is that the relations are heterogeneous: loosening a type $T$ in a term to $S$ requires relating of elements of $[\Gamma]T$ and $[\Gamma]S$. Most clauses of the definition simply lift error approximation pointwise, except for LIO, which exhibits the same asymmetry between $e$ and $e'$ in Theorem 6.1.

The proof of Theorem 6.3 follows the same strategy used for noninterference: we show that the various operations in the semantics preserve $<$, and then argue by composition. This is where it is important to ensure that casts do not modify labels: to prove the correctness of operations with casts, we must ensure that $[T \ll S] \ll [T' \ll S']$ when $T < T'$ and $S < S'$. If the choice of $S$ or $S'$ had an impact on labels in the results, these two functions could not be related.

7 Related work

Gradual Typing and IFC. One of our main inspirations comes from GSLRef [33], a gradual language for fine-grained IFC. GSLRef suggests an intriguing tension between gradual typing and noninterference. In principle, it could have validated the dynamic gradual guarantee by construction, as it is derived from the AGT methodology [33]. However, a direct application of AGT violated noninterference, just like the example in Figure 1 does if we remove the NSU check from $\lambda^{info}$. The solution of GSLRef, unfortunately, was to include an analog of the NSU check that breaks the dynamic gradual guarantee. As hinted in the Introduction, we can witness this failure by adapting the example in Figure 1. The reasons, however, differ slightly from what we’ve seen earlier.

Unlike most dynamic IFC systems, GSLRef does not describe run-time secrecy with single labels, but with intervals of plausible labels. As the program runs, these intervals are refined to rule out labels that invalidate security checks; if they become empty, an error is signaled. This representation, inherited from AGT, allows omitting label annotations entirely from terms and types—a convenient feature for retrofitting IFC to existing programs. Because of the intervals, the checks used by GSLRef to enforce noninterference are more complex than the classic NSU; nevertheless, the gradual guarantee still holds in the program of Figure 1, because the cast induced by the annotation on $b$ ends up modifying the intervals tracked by the program, and thus the result of the NSU analogue.

Rather than adopting GSLRef intervals, GLIO resorts to classic IFC labels and NSU checks. We believe that this choice simplifies the use of first-class labels in a gradual setting, as it is unclear what the semantics of a test $\lambda b : S \Rightarrow S$ should be if $\lambda b : S \Rightarrow S$ returns a set of plausible labels rather than a single label—for instance, the gradual guarantee would force this result to be consistent for all possible program annotations. Moreover, we can recover some of the benefits of label intervals because most values are unlabeled in our coarse-grained discipline, and because we could easily use a default label when allocating references (e.g. the PC label).

As far as we know, GSLRef was the first work to consider the dynamic gradual guarantee for an IFC language. MLGS [12] is an earlier design that predates the guarantee, which it can violate by rewriting the program of Figure 1 to classify data through type casts. Other languages use different interpretations of gradual typing from the one adopted here (which goes back to the criteria of Siek et al. [28]), making it hard to provide analogues of the gradual guarantee, because removing annotations might require adding casts to please the type checker. This behavior appears in the language of Disney and Flanagan [11], which interprets missing labels in types as maximum secrecy, and in LJGS [13].

Dependent Types and IFC. Moving further away from gradual typing, we find designs that use dependent types to make static IFC more flexible, deferring label checks to execution time. This category includes the HLIO Haskell library [10] and Jif [19, 36]. Instead of making the checking of dynamic security levels automatic and guided by the
structure of types, these systems require programmers to manually check the safety of operations that involve dynamic labels. Thanks to first-class labels, our language allows programmers to perform these tests manually, as in the `maybeUpdate` function in Figure 5. However, because of the lack of dependent types, our type system cannot use the information learned from these tests to rule out errors statically. Bridging the gap between these two kinds of analyses is an interesting avenue for future work.

**Gradual Types and Parametricity.** Until recently, the interaction between polymorphism and gradual typing exhibited problems similar to the ones we saw for IFC: there had been several proposals of languages that combine the two features [2, 17, 34], but none of them were able to establish both the dynamic gradual guarantee and parametricity. Indeed, Toro et al. [34] conjectured both properties to be fundamentally incompatible.

To solve this issue, New et al. [21] proposed PolyG\(^\gamma\), a polymorphic calculus based on term-level sealing. In PolyG\(^\gamma\), if we instantiate a polymorphic term \( e : \forall \alpha. X \rightarrow X \) with Int, the result is not of type Int \( \rightarrow \) Int, but rather of type \( X \rightarrow X \), where \( X \) is a fresh sealed type generated during execution. To actually use the instantiated function, the sealed type \( X \) comes with two conversion functions \( \text{seal}_X : \text{Int} \rightarrow X \) and \( \text{unseal}_X : X \rightarrow \text{Int} \); thus, instead of \( e [\text{Int}] + 1 \), as we would write in System F, we would have to write

\[
\text{unseal}_X(e(X \equiv \text{Int})(\text{seal}_X 1)) + 1
\]

for the program to be accepted. PolyG\(^\gamma\) satisfies both the DGG and parametricity; crucially, its DGG does not apply to programs that remove occurrences of seal and unseal, since those live at the term level. Our abandonment of type-guided classification is similar: run-time labels are chosen at the term level, and modifying them falls out of the scope of the DGG. This suggests that future tensions with the DGG might be handled by performing at the term level decisions that in fully static systems are usually left implicit at the type level.

8 Conclusion

We presented GLIO, a gradual IFC type system based on the LIO library [31] that features higher-order functions, general references, coarse-grained IFC, security subtyping and first-class labels. In addition to noninterference, our type system validates the dynamic gradual guarantee, an important correctness criterion for gradual typing. To avoid pitfalls encountered in previous work, we decoupled type annotations from data classification, which our language expresses with typical operations from coarse-grained dynamic IFC.

**Acknowledgments**

The authors would like to thank Eric Tanter, Matias Toro, Justin Hsu and the anonymous reviewers for fruitful discussions and suggestions. This work was supported by NSF award 1704542.

**References**


