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 $\mathrm{GSL_{Ref}}$ language, for example, had to abandon it to validate noninterference.

We show that we can solve this conflict by avoiding 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 $f(\texttt{true})$ 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 $\mathrm{GSL_{Ref}}$ language [33] to enforce noninterference.

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

```
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)
```

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>`.

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 λ^{info} [4], a typical language for dynamic IFC. Values in λ^{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 λ^{info}, `Public`), which can be overridden with `< >`. 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 λ^{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 ($\mathrm{GSL_{Ref}}$ [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 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 noninterference. One possibility would be to modify the semantics of conditionals so that they raise the secrecy of all references that could be updated in either branch [25]. In Figure 1, this would mean raising `y`'s label above `x`'s even when the else branch is taken. Apart from the potential performance 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 programmatically; for instance, we might write `labelOf b == S` 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 return 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 `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 annotations for preventing security errors statically. Following the tradition of gradual typing, GLIO features a notion of *consistent 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

```
let b : Bool<S> = true in labelOf b == S
let b : Bool<P> = true in labelOf b == S
let b = true in labelOf b == S
```

Figure 2. Failure of the DGG with first-class labels and type-guided classification. The first two programs have no reason to fail, and with type-guided classification they terminate successfully with different results. The DGG would force the third program must behave the same way as the first two, which is impossible.

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 $\mathrm{GSL}_{\mathrm{Ref}}$ [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* obviates 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 $\mathrm{GSL}_{\mathrm{Ref}}$ 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-grained IFC, security subtyping and public, first-class labels. 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 satisfies both termination- and error-insensitive noninterference (Section 5) and the gradual guarantee of Siek et al. [28] (Section 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 (statically or dynamically). This category includes λ^{info} [4], Flow

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:

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

```
f :: Lab Bool -> LIO Bool
f x = do
  -- Alternative annotation: Lab[S] Bool
  b :: Lab Bool <- label P True
  b' <- unlabel b
  y  <- new P b'
  z  <- new P b'
  x' <- unlabel x
  if x' then set y False
    else return ()
  y' <- get y
  if y' then set z False
    else return ()
  get z

do { x <- label S True; f x }
```

Figure 3. Translation of the example of Figure 1 into GLIO

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 `lub` 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.

```
h :: Lab[S] Bool -> LIO[P,S] Bool
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 :: Lab[S] Bool -> LIO[P,P] (Ref[S] Bool)
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,[2] 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.)

[2] 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 $\mathrm{GSL_{Ref}}$ [33], which by design would trigger a run-time error, since it treats the last cast as a declassification. This behavior can be replicated in GLIO by replacing the first cast to `Lab[S] Bool` with another call to `label`, as shown in the second program, `labClass`. Classification succeeds, because `S` is more secret than `P`, but the last cast fails for the same reason.

Finally, `refCast` demonstrates the difference between labels for `Ref` and `Lab`. The annotations on reference types fix the labels of their contents, so the final cast to `Ref[S] Bool` fails during execution even though `S` is more secret than `P`. Note that this cast has to come after a cast to the imprecise type `Ref Bool`: were it omitted, the type checker would reject the program, as such a coercion always fails.

3 Language

Having built basic intuition, we are ready for a formal definition. The development assumes a lattice of secrecy labels $l \in L$ ordered by $\preccurlyeq$, comprising joins $\vee$, meets $\wedge$, a bottom element $\bot$ and a top element $\top$. The higher a label, the more confidential the values it classifies, with $\bot$ denoting public data and $\top$ denoting maximum secrecy. A simple choice for L would be a lattice of labels $\{\mathsf{P}, \mathsf{S}\}$ ordered by $\mathsf{P} \preccurlyeq \mathsf{S}$. A more interesting instance is $L = \mathcal{P}(P)$ ordered by the subset relation, where $P = \{\mathsf{Alice}, \mathsf{Bob}, \ldots\}$ is a set of principals that own data, $\bot = \emptyset$, $\top = P$, $\wedge = \cap$ and $\vee = \cup$.

Figure 8 summarizes the syntax of terms and types. To simplify the development, we modify the informal overview of the previous section in two aspects. First, since our main technical challenges pertain to impure code, we conflate pure functions and the LIO monad into a single type $T \xrightarrow{\bar{l}_1, \bar{l}_2} S$, which intuitively corresponds to the type $T \longrightarrow \mathsf{LIO}[\bar{l}_1, \bar{l}_2]\, S$ seen earlier. Because of cast errors, "pure" code in our language still needs to be managed monadically, and this simplification allows us to model pure and impure code with a single monad (cf. Section 4). Second, to allow for a more compact semantics later, we present the syntax in A-normal form [14, 26]: most term formers only allow variables as arguments, and the earlier snippets should be translated into a sequence of let definitions. The first term rows contain usual constructs for manipulating booleans, functions, and the heap. The last rows are specific to IFC, and correspond to the primitives of LIO [31]. Two syntactic forms, new and toLab, take either variables or label constants as arguments to allow for more precise typing rules, as we'll soon see. Type ascription is syntax sugar defined in terms of let, and label is defined in terms of toLab. (Since we don't use a separate monadic type, label and toLab are actually synonyms.)

As usual in gradual languages, the missing annotations in concrete syntax formally correspond to the *gradual label* $? \in \bar{L} \triangleq L \uplus \{?\}$, 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 $\Gamma \vdash_{\bar{l}_1, \bar{l}_2} e : T$ 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 $T' \preccurlyeq T$ 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, $\gamma(\mathsf{Lab}_?(\mathsf{Bool}))$ is $\{\mathsf{Lab}_l(\mathsf{Bool}) \mid l \in L\}$. (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 $T \preccurlyeq S$ holds precisely when there exist $T' \in \gamma(T)$ and $S' \in \gamma(S)$ such that $T' \preccurlyeq S'$, for a suitable subtyping relation $\preccurlyeq$ on fully annotated types. The $\preccurlyeq$ relation on $\bar{L}$, 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 $\vee$ (Figure 11). The definition uses a consistent meet operation $\wedge$ 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

$$\bar{l}_1 \wedge \bar{l}_2 \preccurlyeq \bar{l}_i \preccurlyeq \bar{l}_1 \vee \bar{l}_2 \quad \text{and} \quad T_1 \wedge T_2 \preccurlyeq T_i \preccurlyeq T_1 \vee T_2$$

$$
\begin{aligned}
l &\in L \\
\bar{l} \in \bar{L} &\triangleq L \uplus \{?\} \\
b &\in \{0,1\} \\
c &\in L \uplus \{x, y, z, \ldots\} \\
\oplus &\in \{\wedge, \vee\} \\
\Gamma &\in \mathsf{Var} \rightharpoonup_{\mathrm{fin}} \mathsf{Type} \\
e :: T &\triangleq \mathsf{let}(e, x : T. \ldots) \\
\mathsf{label}(c, x) &\triangleq \mathsf{toLab}(c, x)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Term} \ni e ::= {} & x \mid \mathsf{let}(e_1, x : T.e_2) \mid \mathsf{unit} \mid b \mid \mathsf{if}(x, e_1, e_0) \mid \mathsf{fun}(x :_{\bar{l}} T.e) && \text{standard} \\
& \mid \mathsf{app}(x, y) \mid \mathsf{get}(x) \mid \mathsf{set}(x, y) \mid \mathsf{new}(c, y) \mid \mathsf{eqRef}(x, y) \\
& \mid \mathsf{refLabel}(x) \mid \mathsf{labLabel}(x) \mid \mathsf{pcLabel}() && \text{IFC specific} \\
& \mid l \mid x \oplus y \mid x \preccurlyeq y \mid \mathsf{unlabel}(x) \mid \mathsf{toLab}(c, e) \\
\mathsf{Type} \ni T, S ::= {} & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Label} \mid \mathsf{Ref}_{\bar{l}}(T) \mid \mathsf{Lab}_{\bar{l}}(T) \mid T \xrightarrow{\bar{l}_1, \bar{l}_2} S
\end{aligned}
$$

Figure 8. Syntax of terms and types

$$
\frac{\Gamma(x) = T}{\Gamma \vdash_{\bar{l}, \bar{l}} x : T}
\qquad
\frac{\Gamma \vdash_{\bar{l}_1, \bar{l}_2} e_1 : T' \qquad \Gamma[x \mapsto T] \vdash_{\bar{l}_2, \bar{l}_3} e_2 : S \qquad T' \preccurlyeq T}{\Gamma \vdash_{\bar{l}_1, \bar{l}_3} \mathsf{let}(e_1, x : T.e_2) : S}
\qquad
\frac{}{\Gamma \vdash_{\bar{l}, \bar{l}} \mathsf{unit} : \mathsf{Unit}}
\qquad
\frac{}{\Gamma \vdash_{\bar{l}, \bar{l}} b : \mathsf{Bool}}
$$

$$
\frac{\Gamma(x) = \mathsf{Bool} \qquad \Gamma \vdash_{\bar{l}_1, \bar{l}_2^1} e_1 : T_1 \qquad \Gamma \vdash_{\bar{l}_1, \bar{l}_2^0} e_0 : T_0}{\Gamma \vdash_{\bar{l}_1, \bar{l}_2^1 \vee \bar{l}_2^0} \mathsf{if}(x, e_1, e_0) : T_1 \vee T_0}
\qquad
\frac{\Gamma(x) = \mathsf{Ref}_{\bar{l}}(T)}{\Gamma \vdash_{\bar{l}_1, \bar{l}_1 \vee \bar{l}} \mathsf{get}(x) : T}
$$

$$
\frac{\Gamma(x) = \mathsf{Ref}_{\bar{l}}(T_1) \qquad \Gamma(y) = T_2 \qquad T_2 \preccurlyeq T_1 \qquad \bar{l}_1 \preccurlyeq \bar{l}}{\Gamma \vdash_{\bar{l}_1, \bar{l}_1} \mathsf{set}(x, y) : \mathsf{Unit}}
\qquad
\frac{\Gamma(y) = T \qquad \bar{l}_2 \preccurlyeq l_1}{\Gamma \vdash_{\bar{l}_2, \bar{l}_2} \mathsf{new}(l_1, y) : \mathsf{Ref}_{l_1}(T)}
\qquad
\frac{\Gamma(x) = \mathsf{Label} \qquad \Gamma(y) = T}{\Gamma \vdash_{\bar{l}_2, \bar{l}_2} \mathsf{new}(x, y) : \mathsf{Ref}_{?}(T)}
$$

$$
\frac{\Gamma(x) = \mathsf{Ref}_{\bar{l}_1}(T_1) \qquad \Gamma(y) = \mathsf{Ref}_{\bar{l}_2}(T_2) \qquad T_1 \preccurlyeq T_2 \qquad T_2 \preccurlyeq T_1}{\Gamma \vdash_{\bar{l}, \bar{l}} \mathsf{eqRef}(x, y) : \mathsf{Bool}}
\qquad
\frac{\Gamma[x \mapsto T] \vdash_{\bar{l}_1, \bar{l}_2} e : S}{\Gamma \vdash_{\bar{l}, \bar{l}} \mathsf{fun}(x :_{\bar{l}_1} T.e) : T \xrightarrow{\bar{l}_1, \bar{l}_2} S}
$$

$$
\frac{\Gamma(f) = T_1 \xrightarrow{\bar{l}_2, \bar{l}_3} S \qquad \Gamma(x) = T_2 \qquad T_2 \preccurlyeq T_1 \qquad \bar{l}_1 \preccurlyeq \bar{l}_2}{\Gamma \vdash_{\bar{l}_1, \bar{l}_3} \mathsf{app}(f, x) : S}
\qquad
\frac{\Gamma(x) = \mathsf{Ref}_{\bar{l}}(T)}{\Gamma \vdash_{\bar{l}', \bar{l}'} \mathsf{refLabel}(x) : \mathsf{Label}}
\qquad
\frac{\Gamma(x) = \mathsf{Lab}_{\bar{l}}(T)}{\Gamma \vdash_{\bar{l}', \bar{l}'} \mathsf{labLabel}(x) : \mathsf{Label}}
$$

$$
\frac{}{\Gamma \vdash_{\bar{l}, \bar{l}} \mathsf{pcLabel}() : \mathsf{Label}}
\qquad
\frac{}{\Gamma \vdash_{\bar{l}, \bar{l}} l : \mathsf{Label}}
\qquad
\frac{\Gamma(x) = \Gamma(y) = \mathsf{Label}}{\Gamma \vdash_{\bar{l}, \bar{l}} x \preccurlyeq y : \mathsf{Bool}}
\qquad
\frac{\Gamma(x) = \Gamma(y) = \mathsf{Label}}{\Gamma \vdash_{\bar{l}, \bar{l}} x \oplus y : \mathsf{Label}}
\qquad
\frac{\Gamma(x) = \mathsf{Lab}_{\bar{l}'}(T)}{\Gamma \vdash_{\bar{l}, \bar{l} \vee \bar{l}'} \mathsf{unlabel}(x) : T}
$$

$$
\frac{\Gamma \vdash_{\bar{l}_2, \bar{l}_3} e : T \qquad \bar{l}_3 \preccurlyeq l_1 \vee \bar{l}_2}{\Gamma \vdash_{\bar{l}_2, \bar{l}_2} \mathsf{toLab}(l_1, e) : \mathsf{Lab}_{l_1}(T)}
\qquad
\frac{\Gamma(x) = \mathsf{Label} \qquad \Gamma \vdash_{\bar{l}_2, \bar{l}_3} e : T}{\Gamma \vdash_{\bar{l}_2, \bar{l}_2} \mathsf{toLab}(x, e) : \mathsf{Lab}_{?}(T)}
$$

Figure 9. Typing rules

for $i \in \{1, 2\}$, whenever the result of these operations is defined. Note that when all labels are ?, $T \preccurlyeq S$ 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 $\mathsf{new}(l, -)$ and $\mathsf{toLab}(l, -)$ 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 $\bar{l}_3 \preccurlyeq l_1$ instead of $\bar{l}_3 \preccurlyeq l_1 \vee \bar{l}_2$. 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 $[\![T]\!]$ (Figure 12). As the heap can store arbitrary values, $[\![T]\!]$ contains negative recursive occurrences, which requires some care to handle. To solve this issue, we define $[\![T]\!]$ as a CPO rather than a plain set, by solving a domain equation [30]. We briefly

$$\frac{}{\bar{l} \preccurlyeq ?} \qquad \frac{}{? \preccurlyeq \bar{l}} \qquad \frac{l_1 \preccurlyeq l_2 : L}{l_1 \preccurlyeq l_2 : \bar{L}} \qquad \frac{T \in \{\mathsf{Unit}, \mathsf{Bool}, \mathsf{Label}\}}{T \preccurlyeq T}$$

$$\frac{\bar{l}_1 \preccurlyeq \bar{l}_2 \qquad \bar{l}_2 \preccurlyeq \bar{l}_1 \qquad T_1 \preccurlyeq T_2 \qquad T_2 \preccurlyeq T_1}{\mathsf{Ref}_{\bar{l}_1}(T_1) \preccurlyeq \mathsf{Ref}_{\bar{l}_2}(T_2)}$$

$$\frac{\bar{l}_1 \preccurlyeq \bar{l}_2 \qquad T_1 \preccurlyeq T_2}{\mathsf{Lab}_{\bar{l}_1}(T_1) \preccurlyeq \mathsf{Lab}_{\bar{l}_2}(T_2)}$$

$$\frac{\bar{l}'_1 \preccurlyeq \bar{l}_1 \qquad \bar{l}_2 \preccurlyeq \bar{l}'_2 \qquad T'_1 \preccurlyeq T_1 \qquad T_2 \preccurlyeq T'_2}{T_1 \xrightarrow{\bar{l}_1, \bar{l}_2} T_2 \preccurlyeq T'_1 \xrightarrow{\bar{l}'_1, \bar{l}'_2} T'_2}$$

Figure 10. Consistent subtyping

$$\begin{aligned}
\bar{l} \oplus ? = ? \oplus \bar{l} &\triangleq ? \\
\bar{l} \cap ? = ? \cap \bar{l} &\triangleq \bar{l} \\
l_1 \oplus l_2 &\triangleq l_1 \oplus l_2 \\
\bar{l} \cap \bar{l} &\triangleq \bar{l} \\
\mathsf{Unit} \oplus \mathsf{Unit} &\triangleq \mathsf{Unit} \\
\mathsf{Unit} \cap \mathsf{Unit} &\triangleq \mathsf{Unit} \\
\mathsf{Bool} \oplus \mathsf{Bool} &\triangleq \mathsf{Bool} \\
\mathsf{Bool} \cap \mathsf{Bool} &\triangleq \mathsf{Bool} \\
\mathsf{Label} \oplus \mathsf{Label} &\triangleq \mathsf{Label} \\
\mathsf{Label} \cap \mathsf{Label} &\triangleq \mathsf{Label} \\
\mathsf{Ref}_{\bar{l}_1}(T_1) \oplus \mathsf{Ref}_{\bar{l}_2}(T_2) &\triangleq \mathsf{Ref}_{\bar{l}_1 \cap \bar{l}_2}(T_1 \cap T_2) \\
\mathsf{Ref}_{\bar{l}_1}(T_1) \cap \mathsf{Ref}_{\bar{l}_2}(T_2) &\triangleq \mathsf{Ref}_{\bar{l}_1 \cap \bar{l}_2}(T_1 \cap T_2) \\
\mathsf{Lab}_{\bar{l}_1}(T_1) \oplus \mathsf{Lab}_{\bar{l}_2}(T_2) &\triangleq \mathsf{Lab}_{\bar{l}_1 \oplus \bar{l}_2}(T_1 \oplus T_2) \\
\mathsf{Lab}_{\bar{l}_1}(T_1) \cap \mathsf{Lab}_{\bar{l}_2}(T_2) &\triangleq \mathsf{Lab}_{\bar{l}_1 \cap \bar{l}_2}(T_1 \cap T_2)
\end{aligned}$$

$$\begin{aligned}
&T_1 \xrightarrow{\bar{l}_1, \bar{l}_2} T_2 \oplus T'_1 \xrightarrow{\bar{l}'_1, \bar{l}'_2} T'_2 \\
&\triangleq (T_1 \ominus T'_1) \xrightarrow{\bar{l}_1 \ominus \bar{l}'_1, \bar{l}_2 \oplus \bar{l}'_2} (T_2 \oplus T'_2) \\
&\left(T_1 \xrightarrow{\bar{l}_1, \bar{l}_2} T_2\right) \cap \left(S_1 \xrightarrow{\bar{l}'_1, \bar{l}'_2} S_2\right) \\
&\triangleq (T_1 \cap S_1) \xrightarrow{\bar{l}_1 \cap \bar{l}'_1, \bar{l}_2 \cap \bar{l}'_2} (T_2 \cap S_2)
\end{aligned}$$

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 $\wedge$, and $\ominus$ stands for the other operation.

$$[\![\mathsf{Unit}]\!] \cong 1 \qquad [\![\mathsf{Bool}]\!] \cong 2 \qquad [\![\mathsf{Label}]\!] \cong L$$

$$[\![\mathsf{Ref}_{\bar{l}}(T)]\!] \cong \mathsf{Ref}_{\bar{l}} \qquad [\![\mathsf{Lab}_{\bar{l}}(T)]\!] \cong \mathsf{Lab}_{\bar{l}}([\![T]\!])$$

$$\left[\!\!\left[T \xrightarrow{\bar{l}_1, \bar{l}_2} S\right]\!\!\right] \cong [\![T]\!] \xrightarrow{\text{cont}} \mathsf{LIO}_{\bar{l}_1, \bar{l}_2}([\![S]\!])$$

$$\mathsf{Ref}_{\bar{l}} \triangleq \{(r_n, r_{\mathsf{stamp}}, r_{\mathsf{label}}) \in \mathbb{N} \times L \times \gamma(\bar{l}) \mid r_{\mathsf{stamp}} \preccurlyeq r_{\mathsf{label}}\}$$

$$\mathsf{Lab}_{\bar{l}}(X) \triangleq \{x@l \mid x \in X, l \in \downarrow\bar{l}\}$$

$$\begin{aligned}
\mathsf{LIO}_{\bar{l}_1, \bar{l}_2}(X) \triangleq \{f : \mathsf{Mem} \times \downarrow\bar{l}_1 &\xrightarrow{\text{cont}} \mathsf{Error}(\mathsf{Mem} \times X \times \downarrow\bar{l}_2)_\perp \mid \\
&\forall m_1, l_1, x, m_2, l_2.\ f(m_1, l_1) = (m_2, x, l_2) \Rightarrow \\
&l_1 \preccurlyeq l_2 \wedge \mathsf{valid}(l_1, m_1, m_2)\}
\end{aligned}$$

$$\downarrow\bar{l} \triangleq \{l' \in L \mid l' \preccurlyeq \bar{l}\} \qquad \mathsf{Error}(X) \triangleq X + \{\mathsf{error}\}$$

$$\mathsf{Mem} \triangleq (T : \mathsf{Type}^\circ) \times \mathsf{Ref}_? \rightharpoonup_{\text{fin}} [\![T]\!]$$

$$\mathsf{Type}^\circ \triangleq \{T \in \mathsf{Type} \mid T^\circ = T\}$$

$$\begin{aligned}
&T^\circ \triangleq \\
&\quad T \text{ with all labels replaced by } ?
\end{aligned}$$

$$\begin{aligned}
&\mathsf{valid}(l_1, m_1, m_2) \triangleq \\
&\quad \forall (T, r) \in \mathsf{dom}(m_2). \\
&\qquad l_1 \preccurlyeq r_{\mathsf{label}} \wedge (l_1 \not\preccurlyeq r_{\mathsf{stamp}} \Rightarrow (T, r) \in \mathsf{dom}(m_1))
\end{aligned}$$

Figure 12. Interpretation of types and related constructions on CPOs. To simplify notation, we'll treat the isomorphisms defining $[\![T]\!]$ as equations.

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 $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ have a least upper bound $\bigsqcup_{i \in \mathbb{N}} x_i$. The notation $X \xrightarrow{\text{cont}} Y$ refers to the CPO of continuous functions between X and Y—that is, monotone functions $f : X \to Y$ such that $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$, ordered pointwise. The *lifted CPO* $X_\perp$ extends the CPO X with a least element $\perp$, which represents nontermination. We use equality, or the *discrete order*, on CPOs such as $\mathsf{Ref}_{\bar{l}}$, Type, L and its subsets. $\mathsf{Error}(X)$ is ordered pointwise. The order $m_1 \sqsubseteq m_2$ on Mem holds when $\mathsf{dom}(m_1) = \mathsf{dom}(m_2)$ and $\forall T, r.\ m_1(T, r) \sqsubseteq m_2(T, r)$.

Let us explain these definitions before moving on to the semantics of terms. The CPOs $\mathsf{Lab}_{\bar{l}}(X)$ contain elements of X protected by a dynamic label l; as explained in Section 2, this label is bounded by the annotation $\bar{l}$, not necessarily equal to it. A reference $r = (r_n, r_{\mathsf{stamp}}, r_{\mathsf{label}})$ carries two labels: r_{stamp} corresponds to the PC label at the moment of allocation, and r_{label} corresponds to the secrecy of its contents. As noted in Section 2, r_{label} must exactly match the static annotation

$$[\![\Gamma \vdash_{\bar{l}_1,\bar{l}_2} e : T]\!] : [\![\Gamma]\!] \xrightarrow{\text{cont}} \mathsf{LIO}_{\bar{l}_1,\bar{l}_2}([\![T]\!]) \qquad [\![\Gamma]\!] \triangleq \prod_{x \in \mathsf{dom}(\Gamma)} [\![\Gamma(x)]\!]$$

$$[\![x]\!](s) \triangleq \mathsf{return}(s(x)) \qquad [\![\mathsf{let}(e_1 : T', x : T.e_2)]\!](s) \triangleq \mathsf{do}\begin{cases} v' \leftarrow [\![e_1]\!](s); \\ v \leftarrow [\![T' \preccurlyeq T]\!](v'); \\ [\![e_2]\!](s[x \mapsto v]) \end{cases} \qquad [\![\mathsf{unit}]\!](s) \triangleq \mathsf{return}(1)$$

$$[\![b]\!](s) \triangleq \mathsf{return}(b) \qquad [\![\Gamma \vdash_{\bar{l}_1,\bar{l}_2^1 \vee \bar{l}_2^0} \mathsf{if}(x, e_1, e_0) : T_1 \vee T_0]\!](s) \triangleq \mathsf{do}\begin{cases} b \triangleq s(x) \\ v \leftarrow [\![e_b]\!](s); \\ [\![\bar{l}_2^b \preccurlyeq \bar{l}_2^1 \vee \bar{l}_2^0]\!]; \\ [\![T_b \preccurlyeq T_1 \vee T_0]\!](v) \end{cases}$$

$$[\![\mathsf{get}(x : \mathsf{Ref}_{\bar{l}}(T))]\!](s) \triangleq \mathsf{do}\begin{cases} v \leftarrow \mathsf{get}_{\_,\_,T}(s(x)); \\ [\![T^\circ \preccurlyeq T]\!](v) \end{cases} \qquad [\![\mathsf{set}(x : \mathsf{Ref}_{\bar{l}}(T_1), y : T_2)]\!](s) \triangleq \mathsf{do}\begin{cases} v \leftarrow [\![T_2 \preccurlyeq T_2^\circ]\!](s(y)); \\ \mathsf{set}_{\_,\_,T_2}(s(x), v') \end{cases}$$

$$[\![\mathsf{new}(l_1, y : T)]\!](s) \triangleq \mathsf{do}\begin{cases} v \leftarrow [\![T \preccurlyeq T^\circ]\!](s(y)); \\ \mathsf{new}_{l_1,\_,T}(l_1, v) \end{cases} \qquad [\![\mathsf{new}(x, y : T)]\!](s) \triangleq \mathsf{do}\begin{cases} v \leftarrow [\![T \preccurlyeq T^\circ]\!](s(y)); \\ \mathsf{new}_{?,\_,T}(s(x), v) \end{cases}$$

$$[\![\mathsf{eqRef}(x, y)]\!](s) \triangleq \mathsf{return}(s(x) = s(y)) \qquad [\![\mathsf{fun}(x :_{\bar{l}_1} T.e)]\!](s) \triangleq \mathsf{return}(\lambda v.[\![e]\!](s[x \mapsto v]))$$

$$[\![\Gamma \vdash_{\bar{l}_1,\bar{l}_3} \mathsf{app}(f : T_1 \xrightarrow{\bar{l}_2,\bar{l}_3} S, x : T_2) : S]\!] \triangleq \mathsf{do}\begin{cases} v \leftarrow [\![T_2 \preccurlyeq T_1]\!](s(x)); \\ [\![\bar{l}_1 \preccurlyeq \bar{l}_2]\!]; \\ s(f)(v) \end{cases} \qquad [\![\mathsf{refLabel}(x)]\!](s) \triangleq \mathsf{return}(s(x)_{\mathsf{label}})$$

$$[\![\mathsf{labLabel}(x)]\!](s) \triangleq \mathsf{do}\begin{cases} \_@l \triangleq s(x); \\ \mathsf{return}(l) \end{cases} \qquad [\![\mathsf{pcLabel}()]\!](s)(m, l) \triangleq (\emptyset, l, l) \qquad [\![l]\!](s) \triangleq \mathsf{return}(l)$$

$$[\![x \preccurlyeq y]\!](s) \triangleq \mathsf{return}(s(x) \preccurlyeq s(y)) \qquad [\![x \oplus y]\!](s) \triangleq \mathsf{return}(s(x) \oplus s(y)) \qquad [\![\mathsf{unlabel}(x)]\!](s) \triangleq \mathsf{unlabel}(s(x))$$

$$[\![\mathsf{toLab}(l_1, e)]\!](s) \triangleq \mathsf{toLab}_{l_1,\_,\_,\_}(l_1, [\![e]\!](s)) \qquad [\![\mathsf{toLab}(x, e)]\!](s) \triangleq \mathsf{toLab}_{?,\_,\_,\_}(s(x), [\![e]\!](s))$$

Figure 13. Semantics of typing derivations. The types of some variables and expressions are included for clarity, even though they do not appear in the syntax of terms. Conversely, some of the indices of get, set, new and toLab have been left out, but they can be inferred from the annotations in the corresponding judgments.

on the reference's type, if one is provided. The stamp is not important for program behavior, but it simplifies the proof of noninterference, for reasons that will soon become clear.

We depart from Haskell by following call-by-value rather than call-by-need: functions take forced values as their arguments, rather than elements of a lifted CPO $X_\perp$. This is merely for organizational purposes: call-by-value allows us to segregate divergence as an effect inside LIO, rather than including it explicitly in the denotation of each type.

The CPO $\mathsf{LIO}_{\bar{l}_1,\bar{l}_2}(X)$ corresponds to the computation types of LIO [31] and HLIO [10]. Its elements are functions that take as inputs a memory (Mem) and a PC label ($\downarrow\bar{l}_1$), and that can either run forever ($\perp$), produce an error, or return *memory updates* (Mem), a result (X), and a new PC label ($\downarrow\bar{l}_2$). (Returning updates instead of the final memory is unorthodox, but it simplifies the domain equations [6].) The post-condition on the PC label means that it goes up to track inspected secrets. The post-condition valid, explained next, ensures that memory updates do not leak secrets.

A memory $m \in \mathsf{Mem}$ is a function with finite domain that maps a type T and a reference r to a value $v \in [\![T]\!]$. 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 $\mathsf{valid}(l_1, m_1, m_2)$ describes which memory updates are allowed under the PC label l_1: new and updated locations must pass the NSU check for l_1 ($l_1 \preccurlyeq r_{\mathsf{label}}$), and stamps must reflect their allocation context, which, as hinted earlier, is a technical device to simplify the noninterference proof.

The definition of LIO 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,

$$
\begin{aligned}
\mathsf{unlabel}_{\bar{l}_1,\bar{l}_2,X} &: \mathsf{Lab}_{\bar{l}_2}(X) \longrightarrow \mathsf{LIO}_{\bar{l}_1,\bar{l}_1 \curlyvee \bar{l}_2}(X) \\
\mathsf{unlabel}_{\bar{l}_1,\bar{l}_2,X}(x@l_2)(m,l_1) &\triangleq (\emptyset, x, l_1 \curlyvee l_2) \\
\mathsf{get}_{\bar{l}_1,\bar{l}_2,T} &: \mathsf{Ref}_{\bar{l}_2} \longrightarrow \mathsf{LIO}_{\bar{l}_1,\bar{l}_1 \curlyvee \bar{l}_2}(\llbracket T^\circ \rrbracket) \\
\mathsf{get}_{\bar{l}_1,\bar{l}_2,T}(r)(m,l_1) &\triangleq \begin{cases} (\emptyset, v, l_1 \curlyvee r_{\mathsf{label}}) & \text{if } m(T^\circ, r) = v \\ \mathsf{error} & \text{if } (T^\circ, r) \notin \mathsf{dom}(m) \end{cases} \\
\mathsf{set}_{\bar{l}_1,\bar{l}_2,T} &: \mathsf{Ref}_{\bar{l}_1} \times \llbracket T^\circ \rrbracket \longrightarrow \mathsf{LIO}_{\bar{l}_2,\bar{l}_2}(1) \\
\mathsf{set}_{\bar{l}_1,\bar{l}_2,T}(r,v)(m,l_2) &\triangleq \begin{cases} ([T^\circ, r \mapsto v], 1, l_2) & \text{if } l_2 \preccurlyeq r_{\mathsf{label}} \text{ and } (T^\circ, r) \in \mathsf{dom}(m) \\ \mathsf{error} & \text{otherwise} \end{cases} \\
\mathsf{new}_{\bar{l}_1,\bar{l}_2,T} &: \gamma(\bar{l}_1) \times \llbracket T^\circ \rrbracket \longrightarrow \mathsf{LIO}_{\bar{l}_2,\bar{l}_2}(\mathsf{Ref}_{\bar{l}_1}) \\
\mathsf{new}_{\bar{l}_1,\bar{l}_2,T}(l_1,v)(m,l_2) &\triangleq \begin{cases} ([r \mapsto v], r, l_2) & \text{if } l_2 \preccurlyeq l_1 \text{ and } r = (T^\circ, (n, l_2, l_1)), \text{ with} \\ & n \triangleq \min\{n \mid (T^\circ, (n, l_2, l_1)) \notin \mathsf{dom}(m)\} \\ \mathsf{error} & \text{otherwise} \end{cases} \\
\mathsf{toLab}_{\bar{l}_1,\bar{l}_2,\bar{l}_3,X} &: \gamma(\bar{l}_1) \times \mathsf{LIO}_{\bar{l}_2,\bar{l}_3}(X) \longrightarrow \mathsf{LIO}_{\bar{l}_2,\bar{l}_2}(\mathsf{Lab}_{\bar{l}_1}(X)) \\
\mathsf{toLab}_{\bar{l}_1,\bar{l}_2,\bar{l}_3,X}(l_1,f)(m,l_2) &\triangleq \begin{cases} (m', v@l_1, l_2) & \text{if } f(m,l_2) = (m', v, l_3) \text{ and } l_3 \preccurlyeq l_1 \curlyvee l_2 \\ \mathsf{error} & \text{if } f(m,l_2) = (m', v, l_3) \text{ and } l_3 \not\preccurlyeq l_1 \curlyvee l_2 \\ & \text{or } f(m,l_2) = \mathsf{error} \\ \bot & \text{if } f(m,l_2) = \bot \end{cases}
\end{aligned}
$$

Figure 14. Semantics of typing derivations (continued)

$$
\begin{aligned}
\mathsf{return}_{\bar{l},X} &: X \xrightarrow{\text{cont}} \mathsf{LIO}_{\bar{l},\bar{l}}(X) \\
\mathsf{return}(x)(m,l) &\triangleq (\emptyset, x, l) \\
\mathsf{bind}_{\bar{l}_1,\bar{l}_2,\bar{l}_3,X,Y} &: \mathsf{LIO}_{\bar{l}_1,\bar{l}_2}(X) \times \left(X \xrightarrow{\text{cont}} \mathsf{LIO}_{\bar{l}_2,\bar{l}_3}(Y)\right) \xrightarrow{\text{cont}} \mathsf{LIO}_{\bar{l}_1,\bar{l}_3}(Y) \\
\mathsf{bind}(k,f)(m,l) &\triangleq \begin{cases} (m' \vec{\uplus} m'', y, l'') & \text{if } k(m,l) = (m', x, l') \text{ and} \\ & f(x)(m \vec{\uplus} m', l') = (m'', y, l'') \\ \mathsf{error} & \text{if } k(m,l) = (m', x, l') \text{ and} \\ & f(x)(m \vec{\uplus} m', l') = \mathsf{error} \text{ or } k(m,l) = \mathsf{error} \\ \bot & \text{otherwise} \end{cases} \\
(m \vec{\uplus} m')(r) &\triangleq \begin{cases} m'(r) & \text{if } r \in \mathsf{dom}(m') \\ m(r) & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 15. Monadic operations of LIO

that some memory-related errors are ruled out by the shape of the memory. For instance, if we try to read $m(\mathsf{Bool}, r)$ 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.

$$
\begin{aligned}
[\![T \preccurlyeq S]\!]_{\bar{l}} &: [\![T]\!] \xrightarrow{\text{cont}} \mathsf{LIO}_{\bar{l},\bar{l}}([\![S]\!]) \quad (\text{for } T \preccurlyeq S)\\
[\![T \preccurlyeq T]\!] &\triangleq \mathsf{return} \quad (\text{for } T \in \{\mathsf{Unit}, \mathsf{Bool}, \mathsf{Label}\})\\
[\![\mathsf{Ref}_{\bar{l}_1}(T) \preccurlyeq \mathsf{Ref}_{\bar{l}_2}(S)]\!](n, l_1, l_2) &\triangleq \begin{cases} \mathsf{return}(n, l_1, l_2) & \text{if } l_2 \in \gamma(\bar{l}_2)\\ \lambda(-).\,\mathsf{error} & \text{otherwise}\end{cases}\\
[\![\mathsf{Lab}_{\bar{l}_1}(T_1) \preccurlyeq \mathsf{Lab}_{\bar{l}_2}(T_2)]\!](v@l) &\triangleq \begin{cases} \mathsf{do}\begin{cases} v' \leftarrow [\![T_1 \preccurlyeq T_2]\!](v);\\ \mathsf{return}(v'@l)\end{cases} & \text{if } l \in \downarrow\bar{l}_2\\ \lambda(-).\,\mathsf{error} & \text{otherwise}\end{cases}\\
[\![T \xrightarrow{\bar{l}_1,\bar{l}_2} S \preccurlyeq T' \xrightarrow{\bar{l}_1',\bar{l}_2'} S']\!](f) &\triangleq \mathsf{return}\,\lambda x'.\mathsf{do}\begin{cases} x \leftarrow [\![T' \preccurlyeq T]\!](x');\\ [\![\bar{l}_1' \preccurlyeq \bar{l}_1]\!];\\ y \leftarrow f(x);\\ [\![\bar{l}_2 \preccurlyeq \bar{l}_2']\!];\\ [\![S \preccurlyeq S']\!](y);\end{cases}\\
[\![\bar{l}_1 \preccurlyeq \bar{l}_2]\!] &: \mathsf{LIO}_{\bar{l}_1,\bar{l}_2}(1) \quad (\text{for } \bar{l}_1 \preccurlyeq \bar{l}_2)\\
[\![\bar{l}_1 \preccurlyeq \bar{l}_2]\!](m, l_1) &\triangleq \begin{cases} (\emptyset, 1, l_1) & \text{if } l_1 \in \downarrow\bar{l}_2\\ \mathsf{error} & \text{otherwise}\end{cases}
\end{aligned}
$$

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 $T \preccurlyeq T^\circ$ and $T^\circ \preccurlyeq T$ 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 $\top$).

```
toLab S $ do
  b' <- unlabel b
  if b' then do { new S True; return () }
    else return () }
new S True
```

We can produce a typing judgment $[b \mapsto \mathsf{Lab}_?(\mathsf{Bool})] \vdash_{\bot,\bot} e : \mathsf{Ref}_\top(\mathsf{Bool})$, which corresponds to a function $[\![e]\!]$ of type $\mathsf{Lab}_?(2) \xrightarrow{\text{cont}} \mathsf{LIO}_{\bot,\bot}(\mathsf{Ref}_\top)$. By running this program on two different inputs and an empty memory, we obtain successful executions

$$
\begin{aligned}
[\![e]\!](1@\top)(\emptyset, \bot) &= ([r_0 \mapsto 1, r_1 \mapsto 1], r_1, \bot)\\
[\![e]\!](0@\top)(\emptyset, \bot) &= ([r_1 \mapsto 1], r_1, \bot),
\end{aligned}
$$

where $r_0 = (\mathsf{Bool}, (0, \top, \top))$ is allocated inside the conditional, and $r_1 = (\mathsf{Bool}, (0, \bot, \top))$ 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].

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 $(\approx_l)_{l \in L}$ that characterize what elements of $[\![T]\!]$ are indistinguishable to an observer bounded

CPO	Relation	Definition
$1, 2, L, \mathsf{Ref}_{\bar{l}}$	$x \approx_l y$	$x = y$
$\mathsf{Lab}_{\bar{l}}(X)$	$x_1@l_1 \mathrel{\tilde{\approx}_l} x_2@l_2$	$\forall i \in \{1,2\}.\ l_i \preccurlyeq l \Rightarrow (x_1 \approx_l x_2 \wedge l_1 = l_2)$
	$x_1@l_1 \approx_l x_2@l_2$	$x_1@l_1 \mathrel{\tilde{\approx}_l} x_2@l_2 \wedge l_1 = l_2$
$X \xrightarrow{\text{cont}} Y$	$f \approx_l g$	$\forall x \approx_l y, f(x) \approx_l g(y)$
$\mathsf{LIO}_{\bar{l}_1,\bar{l}_2}(X)$	$f \approx_l g$	$\forall m_1 \approx_l m_2, l', m_1', m_2', x_1, x_2, l_1, l_2.$ $f(m_1, l') = (m_1', x_1, l_1) \wedge g(m_2, l') = (m_2', x_2, l_2)$ $\Rightarrow m_1 \vec{\uplus} m_1' \approx_l m_2 \vec{\uplus} m_2' \wedge x_1@l_1 \mathrel{\tilde{\approx}_l} x_2@l_2$
Mem	$m_1 \approx_l m_2$	$\mathsf{dom}_l(m_1) = \mathsf{dom}_l(m_2) \wedge$ $\forall (T, r) \in \mathsf{dom}(m_1) \cap \mathsf{dom}(m_2).\ m_1(T,r)@r_{\mathsf{label}} \approx_l m_2(T,r)@r_{\mathsf{label}}$
$[\![\Gamma]\!]$	$s_1 \approx_l s_2$	$\forall x \in \mathsf{dom}(\Gamma).\ s_1(x) \approx_l s_2(x)$

$$\mathsf{dom}_l(m) \triangleq \{(T, r) \in \mathsf{dom}(m) \mid r_{\mathsf{stamp}} \preccurlyeq l\}$$

Figure 17. Notions of indistinguishability on CPOs. The definitions assume that the CPOs X and Y carry such notions as well.

by l (Figure 17).[3] 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 $\mathsf{Lab}_{\bar{l}}(X)$: weak ($\tilde{\approx}_l$) and strong ($\approx_l$). Weak indistinguishability is only an auxiliary notion used to define indistinguishability for computations ($\mathsf{LIO}_{\bar{l}_1,\bar{l}_2}(X)$). 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 l can distinguish two memories if they differ either in their sets of low-stamp locations, dom_l, or in two values stored at a low location.

Our goal is to prove $[\![e]\!] \approx_l [\![e]\!]$ for every well-typed program e. This implies that programs do not leak secrets; for example, if $l = \bot$ and $e : \mathsf{Lab}_?(\mathsf{Bool}) \xrightarrow{\bot,\bot} \mathsf{Bool}$, we find that $[\![e]\!](1@\top)(\emptyset, \bot)$ and $[\![e]\!](0@\top)(\emptyset, \bot)$ output the same boolean if both terminate successfully.

Theorem 5.1 (Noninterference). *If $\Gamma \vdash_{\bar{l}_1,\bar{l}_2} e : T$, we have*

$$[\![e]\!] \approx_l [\![e]\!] : [\![\Gamma]\!] \xrightarrow{\text{cont}} \mathsf{LIO}_{\bar{l}_1,\bar{l}_2}([\![T]\!]).$$

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. □

6 Gradual guarantees

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

[3]It would be natural to expect indistinguishability to be decreasing with respect to l: the more power the attacker has, the more can be distinguished. However, this property is not required to prove noninterference, as evidenced by similar proofs in the literature [1, 24].

Labels

$$\frac{l \in L}{l \lhd ?} \qquad \frac{l \in L}{l \lhd l}$$

Types

$$\frac{T \in \{\mathsf{Unit}, \mathsf{Bool}, \mathsf{Label}\}}{T \lhd T} \qquad \frac{\bar{l}_1 \lhd \bar{l}_2 \quad T_1 \lhd T_2}{\mathsf{Ref}_{\bar{l}_1}(T_1) \lhd \mathsf{Ref}_{\bar{l}_2}(T_2)}$$

$$\frac{\bar{l}_1 \lhd \bar{l}_2 \quad T_1 \lhd T_2}{\mathsf{Lab}_{\bar{l}_1}(T_1) \lhd \mathsf{Lab}_{\bar{l}_2}(T_2)}$$

$$\frac{\bar{l}_1 \lhd \bar{l}_1' \quad \bar{l}_2 \lhd \bar{l}_2' \quad T_1 \lhd S_1 \quad T_2 \lhd S_2}{T_1 \xrightarrow{\bar{l}_1,\bar{l}_2} T_2 \lhd S_1 \xrightarrow{\bar{l}_1',\bar{l}_2'} S_2}$$

Environments

$$\frac{\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2) \quad \forall x.\ \Gamma_1(x) \lhd \Gamma_2(x)}{\Gamma_1 \lhd \Gamma_2}$$

Terms

$$\frac{}{e \lhd e} \qquad \frac{e_1 \lhd e_1' \quad T \lhd T' \quad e_2 \lhd e_2'}{\mathsf{let}(e_1, x : T.\ e_2) \lhd \mathsf{let}(e_1', x : T'.\ e_2')}$$

$$\frac{e_1 \lhd e_1' \quad e_2 \lhd e_2'}{\mathsf{if}(x, e_1, e_2) \lhd \mathsf{if}(x, e_1', e_2')} \qquad \frac{\bar{l} \lhd \bar{l}' \quad T \lhd T' \quad e \lhd e'}{\mathsf{fun}(x :_{\bar{l}} T.\ e) \lhd \mathsf{fun}(x :_{\bar{l}'} T'\ e')}$$

$$\frac{e \lhd e'}{\mathsf{toLab}(l, e) \lhd \mathsf{toLab}(l, e')} \qquad \frac{e \lhd e'}{\mathsf{toLab}(x, e) \lhd \mathsf{toLab}(x, e')}$$

Figure 18. Syntactic dynamism relations

can only introduce dynamic type errors, without otherwise changing the behavior of the program.

Theorem 6.1 (Dynamic Gradual Guarantee, Simple)**.** *Suppose that* $e \lhd e'$ *with* $\vdash_{\bot,\bar{l}_2} e : T$ *and* $\vdash_{\bot,\bar{l}_2} e' : T'$.

- *If* $[\![e]\!](\emptyset)(\emptyset,\bot) = \bot$, *then* $[\![e']\!](\emptyset)(\emptyset,\bot) = \bot$.
- *If* $[\![e]\!](\emptyset)(\emptyset,\bot) = (m, v, l)$, *then there exist* m' *and* v' *such that* $[\![e']\!](\emptyset)(\emptyset,\bot) = (m', v', l)$.

The premise $e \lhd e'$, defined on Figure 18, says that e' is obtained from e by replacing some labels on type annotations with ?. 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* $\Gamma \lhd \Gamma'$, $\bar{l}_1 \lhd \bar{l}_1'$, $e \lhd e'$, *and* $\Gamma \vdash_{\bar{l}_1,\bar{l}_2} e : T$, *there exist* $\bar{l}_2' \rhd \bar{l}_2$ *and* $T' \rhd T$ *such that* $\Gamma' \vdash_{\bar{l}_1',\bar{l}_2'} e' : T'$.

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].

Theorem 6.3 (Dynamic Gradual Guarantee, General)**.** *If* $\Gamma \vdash_{\bar{l}_1,\bar{l}_2} e : T$, $\Gamma' \vdash_{\bar{l}_1',\bar{l}_2'} e' : T'$, $\Gamma \lhd \Gamma'$, $\bar{l}_i \lhd \bar{l}_i'$ $(\forall i \in \{1,2\})$, $e \lhd e'$ *and* $T \lhd T'$, *then* $[\![e]\!] \lhd [\![e']\!] : [\![\Gamma]\!] \xrightarrow{\text{cont}} \mathsf{LIO}_{\bar{l}_1,\bar{l}_2}([\![T]\!]) \lhd [\![\Gamma']\!] \xrightarrow{\text{cont}} \mathsf{LIO}_{\bar{l}_1',\bar{l}_2'}([\![T']\!])$.

The *error approximation relations* $[\![e]\!] \lhd [\![e']\!]$ in the conclusion are defined on Figure 19. Like indistinguishability in Section 5, they are constructed using Pitts' 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 $[\![T]\!]$ and $[\![S]\!]$. 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 $\lhd$, 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 $[\![T \preccurlyeq S]\!] \lhd [\![T' \preccurlyeq S']\!]$ when $T \lhd T'$ and $S \lhd S'$. 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 GSL_{Ref} [33], a gradual language for fine-grained IFC. GSL_{Ref} 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 λ^{info}. The solution of GSL_{Ref}, 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, GSL_{Ref} 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 GSL_{Ref} to enforce noninterference are more complex than the classic NSU; nevertheless, the gradual guarantee still breaks 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 GSL_{Ref} 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 `labelOf b == S` should be if `labelOf b` 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, GSL_{Ref} was the first work to consider the dynamic gradual guarantee for an IFC language. ML-GS [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

CPOs	Relation	Definition
$1, 2, L, \mathsf{Ref}_{\bar{l}}$	$x \lhd y$	$x = y$
$\mathsf{Lab}_{\bar{l}}(X)$	$x_1@l_1 \lhd x_2@l_2$	$x_1 \lhd x_2 \wedge l_1 = l_2$
$X \xrightarrow{\text{cont}} Y$	$f \lhd g$	$\forall x \lhd y.\ f(x) \lhd g(y)$
$\mathsf{LIO}_{\bar{l}_1,\bar{l}_2}(X)$	$f \lhd g$	$\forall m_1 \lhd m_2, l.\ (f(m_1, l) = \bot \Rightarrow g(m_2, l) = \bot) \wedge$
		$\forall m_1', x_1', l'.\ f(m_1, l) = (m_1', x_1, l')$
		$\Rightarrow \exists m_2', x_2.\ g(m_2, l) = (m_2', x_2, l') \wedge m_1' \lhd m_2' \wedge x_1 \lhd x_2$
Mem	$m_1 \lhd m_2$	$\mathsf{dom}(m_1) = \mathsf{dom}(m_2) \wedge \forall r \in \mathsf{dom}(m_1).\ m_1(r) \lhd m_2(r)$
$[\![\Gamma]\!]$	$s_1 \lhd s_2$	$\forall x \in \mathsf{dom}(\Gamma), s_1(x) \lhd s_2(x)$

Figure 19. Error approximation on CPOs. The relations are heterogeneous, and the left column should be formally understood as describing pairs of CPOs (e.g. the second row defines a relation $(\lhd_{\bar{l},\bar{l}',X,X'}) \subseteq \mathsf{Lab}_{\bar{l}}(X) \times \mathsf{Lab}_{\bar{l}'}(X')$ in terms of another relation $(\lhd_{X,X'}) \subseteq X \times X'$). We will write $x \lhd y : X \lhd Y$ to indicate the CPOs involved in the relation.

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$^\nu$, a polymorphic calculus based on *term-level sealing*. In PolyG$^\nu$, if we instantiate a polymorphic term $e : \forall^\nu X.\ X \to X$ with Int, the result is not of type $\mathsf{Int} \to \mathsf{Int}$, but rather of type $X \to X$, 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 $\mathsf{seal}_X : \mathsf{Int} \to X$ and $\mathsf{unseal}_X : X \to \mathsf{Int}$; thus, instead of $e\ [\mathsf{Int}]\ 1+1$, as we would write in System F, we would have to write

$$\mathsf{unseal}_X(e\{X \cong \mathsf{Int}\}(\mathsf{seal}_X\ 1)) + 1$$

for the program to be accepted. PolyG$^\nu$ 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 abandon 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 Éric Tanter, Matías Toro, Justin Hsu and the anonymous reviewers for fruitful discussions and suggestions. This work was supported by NSF award 1704542.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *POPL*. ACM, 147–160.

[2] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for free for free: parametricity, with and without types. *PACMPL* 1, ICFP (2017), 39:1–39:28. https://doi.org/10.1145/3110283

[3] Robert Atkey. 2009. Parameterised notions of computation. *J. Funct. Program.* 19, 3-4 (2009), 335–376.

[4] Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security* (Dublin, Ireland) *(PLAS '09)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/1554339.1554353

[5] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. 2016. A verified information-flow architecture. *Journal of Computer Security* 24, 6 (2016), 689–734. https://doi.org/10.3233/JCS-15784

[6] Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. 2020. Reconciling Gradual Type: Full Version with Definitions and Proofs. https://arthuraa.net/docs/glio-full.pdf

[7] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A semantic account of metric preservation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 545–556. http://dl.acm.org/citation.cfm?id=3009890

[8] Anindya Banerjee and David A. Naumann. 2005. Stack-based access control and secure information flow. *J. Funct. Program.* 15, 2 (2005), 131–177.

[9] John Tang Boyland. 2014. The problem of structural type tests in a gradual-typed language. *Foundations of Object-Oriented Langauges* (2014).

[10] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: mixing static and dynamic typing for information-flow control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015.* 289–301. https://doi.org/10.1145/2784731.2784758

[11] Tim Disney and Cormac Flanagan. 2011. Gradual Information Flow Typing. In *Proceedings of the International Workshop on Scripts to Programs.*

[12] Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *CSF.* IEEE Computer Society, 224–239.

[13] Luminous Fennell and Peter Thiemann. 2016. LJGS: Gradual Security Types for Object-Oriented Languages. In *ECOOP (LIPIcs)*, Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 9:1–9:26.

[14] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993.* 237–247. https://doi.org/10.1145/155090.155113

[15] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 429–442. https://doi.org/10.1145/2837614.2837670

[16] Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. 2013. All Your IFCException Are Belong to Us. In *IEEE Symposium on Security and Privacy.* IEEE Computer Society, 3–17.

[17] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On polymorphic gradual typing. *PACMPL* 1, ICFP (2017), 40:1–40:29. https://doi.org/10.1145/3110284

[18] Paul Blain Levy. 2002. Possible World Semantics for General Storage in Call-By-Value. In *CSL (Lecture Notes in Computer Science)*, Vol. 2471. Springer, 232–246.

[19] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *POPL.* ACM, 228–241.

[20] Max S. New and Amal Ahmed. 2018. Graduality from embedding-projection pairs. *PACMPL* 2, ICFP (2018), 73:1–73:30. https://doi.org/10.1145/3236768

[21] Max S. New, Dustin Jamner, and Amal Ahmed. 2020. Graduality and parametricity: together again for the first time. *Proc. ACM Program. Lang.* 4, POPL (2020), 46:1–46:32. https://doi.org/10.1145/3371114

[22] Andrew M. Pitts. 1996. Relational Properties of Domains. *Inf. Comput.* 127, 2 (1996), 66–90.

[23] François Pottier and Vincent Simonet. 2003. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.* 25, 1 (2003), 117–158.

[24] Vineet Rajani and Deepak Garg. 2018. Types for Information Flow Control: Labeling Granularity and Semantic Models. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018.* 233–246. https://doi.org/10.1109/CSF.2018.00024

[25] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010.* 186–199. https://doi.org/10.1109/CSF.2010.20

[26] Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation* 6, 3-4 (1993), 289–360.

[27] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP.* 81–92.

[28] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL (LIPIcs)*, Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 274–293.

[29] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015. Monotonic References for Efficient Gradual Typing. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 432–456. https://doi.org/10.1007/978-3-662-46669-8_18

[30] Michael B. Smyth and Gordon D. Plotkin. 1982. The Category-Theoretic Solution of Recursive Domain Equations. *SIAM J. Comput.* 11, 4 (1982), 761–783.

[31] Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. 2017. Flexible dynamic information flow control in the presence of exceptions. *J. Funct. Program.* 27 (2017), e5. https://doi.org/10.1017/S0956796816000241

[32] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *Haskell.* ACM, 95–106.

[33] Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Trans. Program. Lang. Syst.* 40, 4 (2018), 16:1–16:55. https://dl.acm.org/citation.cfm?id=3229061

[34] Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual parametricity, revisited. *PACMPL* 3, POPL (2019), 17:1–17:30. https://dl.acm.org/citation.cfm?id=3290330

[35] Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. 2019. From fine- to coarse-grained dynamic information flow control and back. *PACMPL* 3, POPL (2019), 76:1–76:31.

[36] Lantian Zheng and Andrew C. Myers. 2007. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.* 6, 2-3 (2007), 67–84.