Reconciling noninterference and gradual typing
Full Version with Definitions and Proofs

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 GSLRef 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.


Keywords: Gradual Typing, Noninterference

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 $\lambda$-calculus of Siek and Taha [26], 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 [28], polymorphism [2, 16, 20, 33], 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) [27]: 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

Recent work has managed to lift this restriction using ideas similar to ours [20]; cf. Section 7.

![Figure 1.](image-url) 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>.
We are led to the second option: abandoning type-guided 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 [24]. 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 [30, 35] 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 [8, 27]: 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 a hint that everything is secure, not a guarantee. Unfortunately, this behavior violates the DGG, because dynamic errors are not allowed to disappear when we provide type annotations.

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. [27] (Section 6). We discuss related work in Section 7 and conclude in Section 8. Detailed proofs and definitions are included in Appendix A.

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.
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 $\lambda^{info}$ [4], Flow Caml [22] and Jif [18], among others. By contrast, systems such as DCC [1], LIO [31] 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 [23, 34], 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 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 [9], 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`.

---

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

```haskell
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 }
```

---

---
\[his \in \text{Lab}[P \times S] \rightarrow LIO[P \times S] \text{Bool}
\]
\[h x = \begin{aligned}
&\text{-- PC label = P} \\
b &\in \text{Lab}[P] \text{Bool} \leftarrow \text{label} P \text{ True} \\
b' &\in \text{Bool} \leftarrow \text{unlabel} b \\
y &\in \text{Ref}[P] \text{Bool} \leftarrow \text{new} P \text{ b'} \\
z &\in \text{Ref}[P] \text{Bool} \leftarrow \text{new} P \text{ b'} \\
x' &\in \text{Bool} \leftarrow \text{unlabel} x \\
&\text{-- PC label = S} \\
\text{if} \ x' \\
&\text{-- Assignment is rejected} \\
&\text{then set} y \text{ False} \\
&\text{else return ()} \\
y &\leftarrow \text{get} y \\
\text{if} \ y \text{ then set} z \text{ False} \\
&\text{else return ()} \\
\text{get} z
\]

\[
\text{do \{} \ x \leftarrow \text{label} S \text{ True}; \ g \ x \ \text{\}}
\]

\textbf{Figure 4.} A fully annotated version of Figure 3 that is rejected at compile time.

\[\text{maybeUpdate} :: \text{Ref} \text{Bool} \rightarrow \text{Lab} \text{Bool} \rightarrow LIO ()
\]
\[\text{maybeUpdate} \ r \ x = \begin{aligned}
&\text{do } \\
\text{lpc} &\leftarrow \text{pclabel} \\
\text{ltx} &\leftarrow \text{lablabel} x \\
\text{lr} &\leftarrow \text{reflabel} r \\
\text{if} \text{ lpc `lub` ltx `canFlowTo` lr then do} \\
\text{x'} &\leftarrow \text{unlabel} x \\
&\text{set} \ r \ x' \\
&\text{else set errorOccurred} \text{ True}
\end{aligned}
\]

\textbf{Figure 5.} Error prevention through label introspection.

\[\text{refLab} :: \text{Lab}[S] \text{Bool} \rightarrow LIO[P \times P] (\text{Ref}[S] \text{Bool})
\]
\[\text{refLab} \ x = \begin{aligned}
&r :: \text{Ref}[S] \text{Bool} \leftarrow \text{new} S \text{ true} \\
&\text{-- toLab :: Label} \rightarrow LIO \text{ a} \rightarrow LIO (\text{Lab} \text{ a}) \\
&\text{toLab} S (\text{do} \{ \ x' \leftarrow \text{unlabel} x; \ \text{set} \ r \ x' \}) \\
&\text{return} \ r
\end{aligned}
\]
\[\text{labRef} :: \text{Ref} \text{Bool} \rightarrow LIO[P \times P] (\text{Lab} \text{Bool})
\]
\[\text{labRef} \ r = \text{toLab} (\text{reflabel} r) \text{ (get} \ r) \]

\[\text{eqRef} :: \text{Ref} \text{Bool} \rightarrow \text{Ref} \text{Bool} \rightarrow LIO[P \times P] \text{Bool}
\]
\[\text{eqRef} \ r1 \ r2 = \text{return} \ (r1 \equiv r2)
\]

\textbf{Figure 6.} Labeling and dynamic allocation.

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.

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. The first argument of toLab is a label that bounds the confidentiality of the result, and its second argument is a computation. If the final PC label after running \(f\) is below 1, toLab wraps the result in a value labeled with 1 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.

\textbf{Casts and classification.} GLIO includes a notion of consistent subtyping to allow annotated and unannotated code.

\textsuperscript{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 [15]. By fixing the final label from the onset, we avoid the issue.
Reconciling noninterference and gradual typing

to interoperate. For example, we may pass a value \( r \) of type
\( \text{Lab Bool} \) to \( \text{refLab} \) 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.)

We can also trigger casts explicitly using type ascription,
as shown in Figure 7. The first function, \( \text{LabCast} \), labels the
boolean \( \text{true} \) with \( P \) and sends it through a series of casts,
indicated with the \( \downarrow : \) operator. The type system checks each
cast to rule out obvious or potential errors, such as coercing
\( \text{Bool} \) to \( \text{Unit} \) or \( \text{Lab}[S] \) \( \text{Bool} \) to \( \text{Lab}[P] \) \( \text{Bool} \).

Once \( \text{LabCast} \) reaches the last cast to \( \text{Lab}[P] \) \( \text{Bool} \), it suc-
cessfully returns \( \text{true} \) 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 constrasts
with previous work [11, 12], in particular with \( \text{GSLRef} \) [32],
which by design would trigger a run-time error, since it treats

\[ \begin{align*}
\text{form} & \quad \text{[13, 25]}: \text{most term formers only allow variables as}
\end{align*} \]

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 \( \text{LIO} \) [30]. 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 \textit{gradual label}
\( ? \in \hat{L} \subseteq L \cup \{?\} \), 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, \( \text{GLIO} \) 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 \( \Gamma \vdash e : T \) correspond to the static annotations
on the \( \text{LIO} \) 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
\( \text{refLabel} \), \( \text{labLabel} \) and \( \text{pcLabel} \).

The rule for \( \text{let} \) shows how the label indices are threaded
through as the computation unfolds. The \textit{consistent subtyp-
ing} assumption \( T' \ll T \) allows weakening security annotations
or even omitting them entirely. Its definition, shown in
Figure 10, resembles the subtyping discipline of Rajani
and Garg [23], but adapted to the gradual setting using the
Abstracting Gradual Typing (AGT) framework [14]. 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
\( \{\text{Lab}(\text{Bool}) | l \in L\} \). (Figure 20 gives the complete defi-
nition.) 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 \( T \ll S \) holds precisely
when there exist \( T' \in \gamma(T) \) and \( S' \in \gamma(S) \) such that \( T' \ll S' \),
for a suitable subtyping relation \( \ll \) on fully annotated types.
The \( \ll \) relation on \( L \), which extends the one on \( L \), can be recast
in the same way, by posing \( \gamma(?) = L \) and \( \gamma(I) = \{I\} \).

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 \textit{partial} consistent join operations \( \nu \) (Figure 11).
The definition uses a consistent meet operation \( \land \) and an
intersection operation \( \land \) on types and gradual labels. These
operations are not joins and meets in the usual sense, since

\[ \begin{align*}
\text{L = P} & \quad \text{[32],}
\end{align*} \]
the consistent orders are not transitive, and thus not actual orders; nevertheless, we can show

\[ l_i \times l_j \times l_k \times l_i \times l_2 \quad \text{and} \quad T_1 \times T_2 \times T_1 \times T_1 \times T_2 \]

for \( i \in \{1, 2\} \), whenever the result of these operations is defined. Note that when all labels are \(?\), \( T = S \) 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 [31], which would translate as \( l_i \sim l_i \) instead of \( l_i \sim l_i \times l_2 \). Intuitively, our variant is sound because the result of toLab is protected by both the ascribed label \( l_i \) and the initial PC label \( l_2 \). In Section 4, we will see that toLab takes the PC label into account during execution too.
Reconciling noninterference and gradual typing

Figure 10. Consistent subtyping

\[ \begin{align*}
\vdash \ l \ \land \ ? & \vdash \ l \\
\vdash \ l \ \lor \ ? & \vdash \ l \\
\vdash \ l \ \land \ l & \vdash \ l \\
\vdash \ l \ \lor \ l & \vdash \ l \\
\text{Unit} \ \land \ \text{Unit} & \vdash \text{Unit} \\
\text{Unit} \ \lor \ \text{Unit} & \vdash \text{Unit} \\
\text{Bool} \ \land \ \text{Bool} & \vdash \text{Bool} \\
\text{Bool} \ \lor \ \text{Bool} & \vdash \text{Bool} \\
\text{Label} \ \land \ \text{Label} & \vdash \text{Label} \\
\text{Label} \ \lor \ \text{Label} & \vdash \text{Label} \\
\text{Ref}_1(T_1) \ \land \ \text{Ref}_1(T_2) & \vdash \text{Ref}_1(T_1 \ \land \ T_2) \\
\text{Ref}_2(T_1) \ \land \ \text{Ref}_2(T_2) & \vdash \text{Ref}_2(T_1 \ \land \ T_2) \\
\text{Lab}_1(T_1) \ \lor \ \text{Lab}_1(T_2) & \vdash \text{Lab}_1(T_1 \ \lor \ T_2) \\
\text{Lab}_2(T_1) \ \lor \ \text{Lab}_2(T_2) & \vdash \text{Lab}_2(T_1 \ \lor \ T_2) \\
T_1 \vdash_{l_1,l_2} T_2 & \vdash (T_1 \ \land \ T_2) \\
\vdash (T_1 \ \lor \ T_2) & \vdash (T_1 \ \lor \ T_2) \\
\vdash (T_1 \ \land \ S_1) & \vdash (T_1 \ \land \ S_1)
\end{align*} \]

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

\[ \begin{align*}
\langle \text{Unit} \rangle & \equiv 1 \\
\langle \text{Bool} \rangle & \equiv 2 \\
\langle \text{Label} \rangle & \equiv L \\
\langle \text{Ref} (T) \rangle & \equiv \text{Ref}_iT \\
\langle \text{Lab} (T) \rangle & \equiv \text{Lab}_iT \\
(T \rightarrow S) & \equiv \text{LIO}_{i_1,i_2} \langle S \rangle \\
\text{Ref}_i & \equiv \{(r_n, r_{\text{stamp}}, r_{\text{label}}) \in \mathbb{N} \times \mathbb{N} \times \gamma(l) \mid r_{\text{stamp}} < r_{\text{label}}\} \\
\text{Lab}_i & \equiv \{x \in l \mid x \in X, l \in \downarrow l\} \\
\text{LIO}_{i_1,i_2} (X) & \equiv \{f : \text{Mem} \times \downarrow l_{i_1} \rightarrow \text{Error} (\text{Mem} \times X \times \downarrow l_{i_2}) \mid \\
\forall m_1, l_1, x, m_2, l_2, f(m_1, l_1) = (m_2, x, l_2) \Rightarrow \\
l_1 \leq l_2 \land \text{valid}(l_1, m_1, m_2)\} \\
\downarrow l & \equiv \{l' \in l \mid l' \leq l\} \\
\text{Error} (X) & \equiv X + \{\text{error}\} \\
\text{Mem} & \equiv (T : \text{Type}') \times \text{Ref}_i \rightarrow_{\text{fin}} \langle T \rangle \\
\text{Type}' & \equiv \{T \in \text{Type} \mid T' = T\} \\
T' & \equiv T \ \land \ \text{with all labels replaced by } ? \ (\text{cf. Figure 21}) \\
\text{valid}(l_1, m_1, m_2) & \equiv \forall (T, r) \in \text{dom}(m_2) \\
l_1 \leq r_{\text{label}} \land (l_1 \neq r_{\text{stamp}} \Rightarrow (T, r) \in \text{dom}(m_1))
\end{align*} \]

Figure 12. Interpretation of types and related constructions on CPOs. To simplify notation, we’ll treat the isomorphisms defining \( \langle T \rangle \) as equations.

4 Semantics

Each type \( T \) in GLIO corresponds to a set \( \langle T \rangle \) (Figure 12). As the heap can store arbitrary values, \( \langle T \rangle \) contains negative recursive occurrences, which requires some care to handle. To solve this issue, we define \( \langle T \rangle \) as a CPO rather than a plain set, by solving a domain equation [29]. We briefly review basic notions needed to cover the main contributions, and postpone a detailed description of the construction to Appendix A for the interested readers.

First, by CPO we mean a partially ordered set where all increasing chains \( x_0 \subseteq x_1 \subseteq \cdots \) have a least upper bound \( \bigvee_{i \in \mathbb{N}} x_i \). 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 \( f(\bigvee_{i \in \mathbb{N}} x_i) = \bigvee_{i \in \mathbb{N}} f(x_i) \), ordered pointwise. The lifted CPO \( X \downarrow \) 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. \( \text{Error} (X) \) is ordered pointwise. The order \( m_1 \subseteq m_2 \) on Mem holds when \( \text{dom}(m_1) = \text{dom}(m_2) \) and \( \forall T, r. m_1 (T, r) \subseteq m_2 (T, r) \).
Let us explain these definitions before moving on to the semantics of terms. The CPOs \(\text{Lab}_X(X)\) 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 \(r = (r_n, r_{\text{stamp}}, r_{\text{label}})\) 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 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 \(X_1\). 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 \(\text{LIO}_{l_1,l_2}(X)\) corresponds to the computation types of LIO [30] and HLIO [9]. Its elements are functions that take \(\langle x, e \rangle\) as inputs and return \(\langle m, l, l \rangle\) and a PC label \((\langle l_1 \rangle, l_1)\). This can either run forever \((\perp)\), produce an error, or return memory updates \((\text{Mem}, \text{Mem})\), a result \((X)\), and a new PC label \((\langle l_2 \rangle, l_2)\). (Returning updates instead of the final memory is unorthodox, but it simplifies the domain equations, as discussed in Appendix A.2.) 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 \(m \in \text{Mem}\) is a function with finite domain \(\text{dom}(m)\), \(\text{Mem}\) describes which memory updates are valid \(\cup \), \(\cup \) describes which memory updates are valid \(\cup \). This is a more efficient approach below). The predicate valid \((l_1, m_1, m_2)\) describes which memory updates are
Reconciling noninterference and gradual typing

\[ \text{unlabel}_{i, j, X} : \text{Lab}_i(X) \rightarrow \text{LIO}_{i, j, \cdot, \cdot}(X) \]
\[ \text{unlabel}_{i, j, X}(x@l_2)(m, l_1) \equiv (\emptyset, x, l_1 \lor l_2) \]
\[ \text{get}_{i, j, T} : \text{Ref}_i \rightarrow \text{LIO}_{i_1, j_1, \cdot, \cdot}([T]) \]
\[ \text{get}_{i, j, T}(r)(m, l_1) \equiv \begin{cases} (\emptyset, v, l_1 \lor \eta_{\text{label}}) & \text{if } (T, r) = v \\ \text{error} & \text{if } (T, r) \notin \text{dom}(m) \end{cases} \]
\[ \text{set}_{i, j, T} : \text{Ref}_i \times \text{LIO}_{i_1, j_1, \cdot, \cdot}([T]) \rightarrow \text{LIO}_{i_1, j_1, \cdot, \cdot}(1) \]
\[ \text{set}_{i, j, T}(r, v)(m, l_2) \equiv \begin{cases} ([T, r \mapsto v], 1, l_2) & \text{if } l_2 \ll \eta_{\text{label}} \text{ and } (T, r) \in \text{dom}(m) \\ \text{error} & \text{otherwise} \end{cases} \]
\[ \text{new}_{i, j, T} : \eta_{\text{LLO}} \rightarrow \text{LIO}_{i_1, j_1, \cdot, \cdot}(\text{Ref}_i) \]
\[ \text{new}_{i, j, T}(l_1, v)(m, l_2) \equiv \begin{cases} ([r \mapsto v], r, l_2) & \text{if } l_2 \ll l_1 \text{ and } r = (T, (n, l_2, l_1)), \text{ with} \\ & n \equiv \min\{n \mid (T, (n, l_2, l_1)) \notin \text{dom}(m)\} \\ \text{error} & \text{otherwise} \end{cases} \]
\[ \text{toLab}_{i, j, i, j, X} : \eta_{\text{Lab}} \times \text{LIO}_{i_1, j_1, \cdot, \cdot}(X) \rightarrow \text{LIO}_{i_1, j_1, \cdot, \cdot}(\text{Lab}_i(X)) \]
\[ \text{toLab}_{i, j, i, j, X}(l_1, v)(m, l_2) \equiv \begin{cases} (m', v@l_1, l_2) & \text{if } f(m, l_2) = (m', v, l_3) \text{ and } l_3 \ll l_1 \lor l_2 \\ \text{error} & \text{if } f(m, l_2) = (m', v, l_3) \text{ and } l_3 \not\ll l_1 \lor l_2 \\ & \text{or } f(m, l_2) = \text{error} \\ \bot & \text{if } f(m, l_2) = \bot \end{cases} \]

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

\[ \text{return}_{L, X} : X \xrightarrow{\text{cont}} \text{LIO}_{i, j, \cdot, \cdot}(X) \]
\[ \text{return}(x)(m, l) \equiv (\emptyset, x, l) \]
\[ \text{bind}_{i_1, j_2, i_2, j_2, X, Y} : \text{LIO}_{i_1, j_2}(X) \times \left( X \xrightarrow{\text{cont}} \text{LIO}_{i_2, j_1}(Y) \right) \xrightarrow{\text{cont}} \text{LIO}_{i_1, j_2}(Y) \]
\[ \left( m' \triangleright m'', y, l'' \right) \text{ if } k(m, l) = (m', x, l') \text{ and } f(x)(m' \triangleright m', l') = (m'', y, l'') \]
\[ \text{bind}(k, f)(m, l) \equiv \begin{cases} \text{error} & \text{if } k(m, l) = (m', x, l') \text{ and } f(x)(m' \triangleright m', l') = \text{error} \text{ or } k(m, l) = \text{error} \\ \bot & \text{otherwise} \end{cases} \]
\[ (m \triangleright m')(r) \equiv \begin{cases} m'(r) & \text{if } r \notin \text{dom}(m') \\ m(r) & \text{otherwise} \end{cases} \]

**Figure 15.** Monadic operations of LIO

allowed under the PC label \( l_1 : \text{new and updated locations} \) must pass the NSU check for \( l_1 (l_1 \ll \eta_{\text{label}}) \), and stamps must reflect their allocation context, which, as hinted earlier, is a technical device to simplify the noninterference proof.

The definition of 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 [17], 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
we use a cast to forget the labels. (Note that (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.

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 $T \ll T'$ and $T \ll T$ hold for every $T$.) A more efficient approach would be to use monotonic references [28], 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 [30], but toLab includes the initial PC label $\tilde{l}_1$ 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 $\top$).

```haskell
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 $[b \mapsto \text{Lab}_{T}(\text{Bool})] \vdash \perp \perp e : \text{Ref}_{T}(\text{Bool})$, which corresponds to a function $[e]$ of type $\text{Lab}_{T}(2) \longrightarrow \text{LIO}_{\perp \perp}(\text{Ref}_{T})$. By running this program on two different inputs and an empty memory, we obtain successful executions

$[e](1@\top)(\emptyset, \perp) = ([r_0 \mapsto 1, r_1 \mapsto 1], r_1, \perp)$

$[e](0@\top)(\emptyset, \perp) = ([r_1 \mapsto 1], r_1, \perp)$

where $r_0 = (\text{Bool}, (0, 0, \top))$ 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 [7].

![Figure 16. Label and type coercion](image-url)
Reconciling noninterference and gradual typing

<table>
<thead>
<tr>
<th>CPO</th>
<th>Relation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1, 2, \text{Ref}$</td>
<td>$x \equiv y$</td>
<td>$x = y$</td>
</tr>
<tr>
<td>Lab$(X)$</td>
<td>$x_1@_l \equiv x_2@_l ; \forall l \in {1, 2}$</td>
<td>$(x_1 = x_2 \land l_1 = l_2)$</td>
</tr>
<tr>
<td>$X \xrightarrow{\text{cont}} Y$</td>
<td>$f \equiv g$</td>
<td>$\forall x \equiv y, f(x) \equiv g(y)$</td>
</tr>
<tr>
<td>LIo$_{l_1,l_2}(X)$</td>
<td>$f \equiv g$</td>
<td>$\forall m_1 \equiv m_2, T', m_1', m_2', x_1, x_2, l_1, l_2,$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$f(m_1, l') = (m_1', x_1, l_1) \land g(m_2, l') = (m_2', x_2, l_2),$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Rightarrow m_1 \equiv m_2 \land m_1' \equiv m_2' \land x_1 @ l_1 \equiv x_2 @ l_2$</td>
</tr>
<tr>
<td>Mem</td>
<td>$m_1 \equiv m_2$</td>
<td>dom$(m_1) = \text{dom}(m_2) \land$</td>
</tr>
<tr>
<td>[Γ]</td>
<td>$s_1 \equiv s_2$</td>
<td>$\forall x \in \text{dom}(Γ). s_1(x) \equiv s_2(x)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{dom}(m) = {(T, r) \mid r_{\text{stamp}} \equiv l}$</td>
</tr>
</tbody>
</table>

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

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 $(\equiv_l)_l$, that characterize what elements of $[T]$ are indistinguishable to an observer bounded by $l$ (Figure 17). The definition is again circular, but it can be solved with Pitts’ framework of relational structures [6, 21], as explained in Appendix A.3.

For base types and references, being indistinguishable simply means being equal. There are two notions of indistinguishability for Lab$(X)$: weak ($\equiv_l$) and strong ($\equiv_l$). Weak indistinguishability is only an auxiliary notion used to define indistinguishability for computations (LIo$_{l_1,l_2}(X)$). 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 $\llbracket e \rrbracket \equiv_l \llbracket e' \rrbracket$ for every well-typed program $e$. This implies that programs do not leak secrets; for example, if $l = \bot$ and $e : \text{Lab}(\text{Bool}) \xrightarrow{\bot} \text{Bool}$, we find that $\llbracket e \rrbracket(1@T)(\emptyset, l)$ and $\llbracket e' \rrbracket(0@T)(\emptyset, l)$ output the same boolean if both terminate successfully.

Theorem 5.1 (Noninterference). If $\Gamma \vdash_{l_1,l_2} e : T$, we have

\[
\llbracket e \rrbracket \equiv_l \llbracket e' \rrbracket : [\Gamma] \xrightarrow{\text{cont}} \text{LIo}_{l_1,l_2}([T]).
\]

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. The details are discussed in Appendix A.3.

6 Gradual guarantees

The main novelty of GLIO is that it satisfies the dynamic gradual guarantee [27]: making label annotations more precise can only introduce dynamic type errors, without otherwise changing the behavior of the program.

Theorem 6.1 (Dynamic Gradual Guarantee, Simple). Suppose that $e \equiv e'$ with $\vdash_{-\bot,l_2} e : T$ and $\vdash_{-\bot,l_2} e' : T'$.

- If $\llbracket e \rrbracket(\emptyset) = \bot$, then $\llbracket e' \rrbracket(\emptyset) = \bot$.
- If $\llbracket e \rrbracket(\emptyset) = (m, v, l)$, then there exist $m'$ and $v'$ such that $\llbracket e' \rrbracket(\emptyset) = (m', v', l)$.

The premise $e \equiv e'$, 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 $\Gamma \equiv \Gamma'$, $l_1 \equiv l_1'$, $e \equiv e'$, and $\Gamma \vdash_{l_1,l_2} e : T$, there exist $l_2' \equiv l_2$ and $T' \equiv T$ such that $\Gamma' \vdash_{l_1,l_2'} e' : T'$.

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 [19, 20].

\footnote{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, 23].}
that the various operations in the semantics preserve \( \preceq \), 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 \([T \preceq S] \preceq [T' \preceq S']\) when \(T \sim T'\) and \(S \sim S'\). 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 [32], 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 [32]. 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 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 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 \(1b\text{e}10f b == S\) should be if \(1b\text{e}10f b\) 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, GSLRef was the first work to combine IFC and gradual typing and noninterference. We believe that it could have solved the problem of noninterference in gradual typing, but 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.

The error approximation relations \([e] \preceq [e']\) in the conclusion are defined on Figure 19. Like indistinguishability in Section 5, they are constructed using Pitts’ work [6, 21] (cf. Appendix A.4). 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 LIQ, which exhibits the same asymmetry between \(e\) and \(e'\) in Theorem 6.1.

The proof of Theorem 6.3, detailed in Appendix A.4, follows the same strategy used for noninterference: we show
Reconciling noninterference and gradual typing

<table>
<thead>
<tr>
<th>CPOs</th>
<th>Relation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, L, Ref</td>
<td>$x \ni y$</td>
<td>$x = y$</td>
</tr>
<tr>
<td>Lab(X)</td>
<td>$x_1 \ni_1 l_1 \ni_2 x_2 \land l_1 = l_2$</td>
<td>$x_1 = x_2 \land l_1 = l_2$</td>
</tr>
<tr>
<td>X $\xrightarrow{\ni} Y$</td>
<td>$f \ni g$</td>
<td>$\forall x \ni y. f(x) \ni g(y)$</td>
</tr>
<tr>
<td>LIO_{l, l}(X)</td>
<td>$f \ni g$</td>
<td>$\forall m_1 \ni m_2, l. \langle f(m_1, l) = \bot \Rightarrow g(m_2, l) = \bot \rangle \land \forall m_1, x_1, l'. f(m_1, l) = (m_1, x_1, l') \Rightarrow \exists m_2', x_2. g(m_2, l) = (m_2', x_2, l') \land m_1' \ni m_2' \land x_1 = x_2$</td>
</tr>
<tr>
<td>Mem</td>
<td>$m_1 = m_2$</td>
<td>$\forall r \in \text{dom}(m_1). m_1(r) = m_2(r)$</td>
</tr>
<tr>
<td>$[I]$</td>
<td>$s_1 = s_2$</td>
<td>$\forall x \in \text{dom}(I), s_1(x) = s_2(x)$</td>
</tr>
</tbody>
</table>

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 $(\ni_{L^P, X, X'}) \subseteq \text{Lab}(X) \times \text{Lab}(X')$ in terms of another relation $(\ni_{X, X'}) \subseteq X \times X'$). We will write $x \ni y : X \ni Y$ to indicate the CPOs involved in the relation.

here (which goes back to the criteria of Siek et al. [27]), 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 [10], which interprets missing labels in types as maximum secrecy, and in LJGS [12].

Dependent Types and IFC. Moving further away from gradual typing, we find that designs use dependent types to make static IFC more flexible, deferring label checks to execution time. This category includes the HLIO Haskell library [9] and Jif [18, 35]. 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, 16, 33], but none of them were able to establish both the dynamic gradual guarantee and parametricity. Indeed, Toró et al. [33] conjectured both properties to be fundamentally incompatible.

To solve this issue, New et al. [20] proposed PolyG*, a polymorphic calculus based on term-level sealing. In PolyG*, if we instantiate a polymorphic term $e : \forall X. X \rightarrow X$ with Int, the result is not of type $\text{Int} \rightarrow \text{Int}$, 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 $e[\text{Int}] + 1 + 1$, 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* 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 [30] 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, Matías Toro, Justin Hsu and the anonymous reviewers for fruitful discussions and suggestions. This work was supported by NSF award 1704542.

References
A Detailed proofs and definitions

We detail here definitions and proofs related to the type system, the semantics of types (Section 4), noninterference (Section 5) and the dynamic gradual guarantee (Section 6).

A.1 Auxiliary Definitions

Figure 20 defines the concretization functions $\gamma$ that interpret gradual types as sets of annotated types. To define the memory in GLIO, we use a type erase operation $T^*$, which
We express the domain equation using a mixed-variance A.2 Defining the interpretation domains

objectsis shown on Figure 22. Here, functors domain equation with the method of Smyth and Plotkin [29]. The CPOs used to interpret types are defined by solving a for all $T, S \in \text{Type}$, let $\gamma(T)$ denote the interpretation of $T$ in the domain equation, which is obtained by composing the actions of simpler pieces: $(-) \xrightarrow{\gamma} (-)_{\bot}$, MemF, etc. If instead we interpreted the result as being the final memory, we would have to modify LIOF to require that the contents of certain memory locations remain unchanged in the result, which would be inconvenient because the standard action of the functor that maps $(X^-, X^+, Y)$ to $\text{MemF}(X^-) \times \downarrow l_1 \xrightarrow{\text{cont}} \text{Error}(\text{MemF}(X^+) \times Y \times \downarrow l_2)_{\bot}$ does not preserve this property.

A.3 Proving noninterference
To define the indistinguishability relations $\approx_{l}$ of Section 5, we apply the method of Pitts [21], following the formulation of Azevedo de Amorim et al. [6]. We define a CLat_{\ast}-fibration $q : \text{Ind} \rightarrow \text{CPO}_\bot^{\text{Type}}$ by a change of base:

\[
\begin{array}{ccc}
\text{Ind} & \xrightarrow{\text{Adm}_{\ast}^{\text{Type} \times L}} & \text{Adm} \\
\downarrow q & & \downarrow p \\
\text{CPO}_\bot^{\text{Type}} & \xrightarrow{f} & \text{CPO}_\bot^{\text{Type} \times L}
\end{array}
\]

where $p : \text{Adm} \rightarrow \text{CPO}_\bot$ is the CLat_{\ast}-fibration of subsets of CPOs that are closed under limits of chains (so-called admissible subsets), and $f(X(T, l) \triangleright X(T) \times X(T)$). This means that each object of Ind can be seen as a pair $(X, R)$, where $X \in \text{CPO}_\bot^{\text{Type}}$ and $(R(T, l) \subseteq X(T) \times X(X))_{T \in \text{Type}, l \in L}$ is a family of relations closed under limits of chains. Moreover, $q$ is admissible, in the sense that Ind is canonically a CPO-category derived from $\text{Adm}_{\ast}^{\text{Type} \times L}$, and this structure is preserved by $q$. We can lift $F$ to a functor $\tilde{F}$ on Ind:

\[
\begin{array}{ccc}
\text{Ind}^{op} \times \text{Ind} & \xrightarrow{\tilde{F}} & \text{Ind} \\
\downarrow \gamma \circ \phi \times q & & \downarrow q \\
\left(\text{CPO}_\bot^{\text{Type}}\right)^{op} \times \text{CPO}_\bot^{\text{Type}} & \xrightarrow{F} & \text{CPO}_\bot^{\text{Type}}
\end{array}
\]
whose action on objects is given by
\[
\tilde{F}^*((X^-, R^-), (X^+, R^+))(T, I)
\]
\[\doteq (F(X^-, X^+)(T), \tilde{F}^R(R^-, R^+)(T, I)),\]
where the lifting maps \(\tilde{F}^R\) are defined in Figure 23. The existence of this lifting means in particular that \(\tilde{F}^R\) depends covariantly on \(R^+\) and contravariantly on \(R^-\) and that it is admissible. Roughly, the last condition holds because the predicates involved in the definition either mention discrete CPOs, for which all sets are admissible, or relations that are assumed to be admissible.

By the aforementioned results, we can construct a family of relations \((R^+_T(T, I) \in \mathcal{D}(T) \times \mathcal{D}(T))_{T \in \text{Type}, L \in \text{EL}}\) such that
\[
(F(x), f(y)) \in R^+_T(T, I)
\]
\[\Doteq (x, y) \in \tilde{F}^R(R_D, R_D)(T, I).\]

Given a type \(T\) and elements \(x, y \in \llbracket T \rrbracket\), we pose \(x \equiv_T y\) if \((x, y) \in R^+_T(T, I)\), and define similar notations for the auxiliary relations \(G^+\) of Figure 23. Modulo the fold isomorphisms, this mostly matches the definitions given in Figure 17, except for a slight gap between the definition of \(\equiv_T\) for LIO in Figure 17 and the relations \(G^\text{LIO}\) of Figure 23: the conclusion of the latter includes the input memories \(m_1\) and \(m_2\), which are absent in the former. Nevertheless, we can show that the two formulations are equivalent. We begin with the following auxiliary result, which says that indistinguishability is trivial when the initial PC label is high.

**Lemma A.2.** For all \(f, g \in \text{LIO}_{l_1, l_1'}\), suppose that
\[
f(m_1, l_1) = (m_1', x_1, l_1') \quad \text{and} \quad g(m_2, l_2) = (m_2', x_2, l_2').
\]
If \(m_1 \equiv m_2\) and \(l_1 \equiv l_2\) for all \(i\), then \(m_1 \equiv m_2\) and \(x_1 \equiv x_2\).

**Proof.** Note that \(l'_i \equiv l_i\) for all \(i\) by the definition of LIO. Hence, \(x_1 \equiv x_2\) vacuously. It remains to show that \(m_1 \equiv m_2\).

We first prove that \(\text{dom}(m_1 \equiv m_2') = \text{dom}(m_2 \equiv m_2')\). Since \(\text{dom}(m_1) = \text{dom}(m_2)\) and for all \(i\) we have \(\text{dom}(m_1) \equiv \text{dom}(m_2)\), it suffices to show
\[
\text{dom}(m_1) \subseteq \text{dom}(m_2).
\]

for all \(i\). If \((T, r) \in \text{dom}(m_2)\) and \(r_{\text{stamp}} \equiv l\), we must have \(l \equiv r_{\text{stamp}}\), since \(l \equiv r_{\text{stamp}}\) would imply the contradiction \(l \not\equiv l\). In this case, the definition of LIO guarantees that \((T, r) \in \text{dom}(m_2)\), and hence \((T, r) \in \text{dom}(m_2)\).

To conclude, we must show
\[
(m_1 \equiv m_2')(T, r) @ r_{\text{label}} = (m_2 \equiv m_2')(T, r) @ r_{\text{label}}
\]
for all \(T\) and for all \(r\), whenever both sides are defined. If \(r_{\text{label}} \not\equiv l\), this is trivial. Otherwise, we must have \((m_1 \equiv m_2')(T, r) = m_1(T, r)\) for all \(i\) if \(r_{\text{label}} \equiv l\) and \(r \in \text{dom}(m_1)\), the definition of LIO applied to \(f\) implies that \(l \equiv r_{\text{label}}\), hence \(l \equiv l\), a contradiction. We conclude because \(m_1 \equiv m_2\).
Figure 23. Relational lifting of $F$ for indistinguishability, along auxiliary definitions. Note that the $R$ parameter of $G^e_{Lab}$ is a single relation, and not a family of relations.

Lemma A.3. Given $f, g \in \text{LIO}_1 \times_1 (X)$, we have $f \equiv_1 g$ in the sense of Figure 17 if and only if $(f, g) \in G^e_{\text{LIO}}(R^e_0, R^e_0, I)$; that is, when $f$ and $g$ satisfy

$$\forall m_1 = m_2, l' \neq l, m'_1, m'_2, x_1, x_2, l_1, l_2.$$

$$f(m_1, l') = (m'_1, x_1, l_1) \land g(m_2, l') = (m'_2, x_2, l_2) \Rightarrow \text{dom}(m_1 \uplus m'_1) = \text{dom}(m_2 \uplus m'_2) \land m'_1 = m'_2 \land x_1 \equiv_1 x_2 \equiv_1 l_1 \equiv_1 l_2.$$

Proof. Write $f =_1 g$ for $(f, g) \in G^e_{\text{LIO}}(R^e_0, R^e_0, I)$ and $m_1 =_1 m_2$ for $(m_1, m_2) \in G^e_{\text{Mem}}(R^e_0, I)$. Note that $m_1 =_1 m_2 \iff \text{dom}(m_1) = \text{dom}(m_2) \land m_1 =_1 m_2$.

$(\Rightarrow)$ When all the premises of $f =_1 g$ are satisfied, we can apply the hypothesis $f =_1 g$ to conclude $m_1 \uplus m'_1 =_1 m_2 \uplus m'_2$ and $x_1 \equiv_1 x_2 \equiv_1 l_1 \equiv_1 l_2$. It suffices to show that $m'_1 =_1 m'_2$. Suppose that we have $(T, r) \in \text{dom}(m'_1) \cap \text{dom}(m'_2)$. This implies $(T, r) \in \text{dom}(m_1 \uplus m'_1) \cap \text{dom}(m_2 \uplus m'_2)$, which yields, thanks to the above hypothesis,

$$m'_1(T, r)@\text{r_\text{label}} = (m_1 \uplus m'_1)(T, r)@\text{r_\text{label}} = x_1 \equiv l_1 \equiv_1 x_2 \equiv l_2.$$

$(\Leftarrow)$ Suppose we have values that satisfy the premises of $f =_1 g$. There are two cases to consider. If $l' \neq l$, it suffices to apply Lemma A.2. Otherwise, if $l' \leq l$, we can apply the hypothesis $f =_1 g$ and conclude $\text{dom}(m_1 \uplus m'_1) = \text{dom}(m_2 \uplus m'_2)$, $m'_1 =_1 m'_2$, and $x_1 \equiv l_1 \equiv_1 x_2 \equiv l_2$. We conclude by noting that $=_1$ is stable under $\uplus$, so that $m_1 \uplus m'_1 =_1 m_2 \uplus m'_2$, and thus $m_1 \uplus m'_1 =_1 m_2 \uplus m'_2$.

We are now ready to proceed with the proof of noninterference. We begin with a few auxiliary lemmas about the indistinguishability relation.

Lemma A.4. Let $=_l$ denote one of $=_1$ or $=_2$.

1. $x@l' = y@l' \iff x@l' = y@l'$
2. If $x =_1 y$, then $x@l' =_1 y@l'$ for any $l'$.
3. If $x@l_v = y@l_v$, then $x@((l_v \land l'_v) =_1 y@((l_v \land l'_v)$.
4. If $x@l_v =_1 y@l_v'$, then either $l_v = 0$ or $l_v = 1$. If $x =_1 y$ or $l_v < l$ and $l_v < l'$.

Proof.

1. By definition, the first relation is just $x@l' =_1 y@l'$.
2. By the previous item, it suffices to show $x@l' =_1 y@l'$. Unfolding definitions, this means showing that $l' \leq l$ implies $x =_1 y$ and $l' \leq l'$, which follows from the assumption.

3. First, suppose that $=_1$ is $=_1$. Assume that one of $l_v \land l'_v$ or $l_v \land l'_v$ is below $l$. This implies that one of $l_v$ or $l_v$ is below $l$. From the assumption $x@l_v =_1 y@l_v$, we find that $x =_1 y$ and $l_v = l_v$. We conclude by appealing to the previous item.

Next, suppose that $=_1$ is $=_1$. By definition, we have $x@l_v =_1 y@l_v$ and $l_v = l_v$. The previous sub-proof shows $x@((l_v \land l'_v) =_1 y@((l_v \land l'_v)$, which implies $x@((l_v \land l'_v) =_1 y@((l_v \land l'_v)$ by definition.

Either $l_v < l$ or $l_v = l$ or $l_v < l$. In the latter case, we are done. In the former case, the definition of $=_1$ implies $x =_1 y$ and $l_v = l_v$, allowing us to conclude.

Lemma A.5. Indistinguishability on memories satisfies the following properties.

1. $\emptyset =_1 \emptyset : \text{Mem}$
2. If $v_1 =_1 v_2$, then $[T, r \mapsto v_1] =_1 [T, r \mapsto v_2]$
3. If \( m_1 =_1 m_2 : \text{Mem} \) and \( m_1' =_1 m_2' : \text{Mem} \), then 
\( m_1 \uplus m_1' =_1 m_2 \uplus m_2' : \text{Mem} \).

**Proof.** Trivial; the domains are empty, so there are no locations to relate.

2. Since the domains on both sides are equal to \( \{(T, r)\} \), we just have to show that \( v_i @ \rho_{\text{label}} =_1 v_j @ \rho_{\text{label}} \). This follows from the assumption and from Lemma A.4.

3. Let \( m''_i = m_i \uplus m_i' \) for \( i \in \{1, 2\} \). We first need to show that the public domains are equal: \( \text{dom}(m''_i) = \text{dom}(m_i') \). This follows because 
\[
\begin{align*}
\text{dom}(m''_i) &= \text{dom}(m_i) \cup \text{dom}(m_i') \\
&= \text{dom}(m_2) \cup \text{dom}(m_2') \quad \text{(by assumption)} \\
&= \text{dom}(m_2'').
\end{align*}
\]

Now, suppose that we have some \( T \) and some \( r \) such that \( (T, r) \in \text{dom}(m''_i) \cap \text{dom}(m''_j) \). We need to show 
\( m''_i(T, r) @ \rho_{\text{label}} =_1 m''_j(T, r) @ \rho_{\text{label}}. \)

Thus, assume \( \rho_{\text{label}} \prec l \). The definition of references implies that \( r_{\text{stamp}} \prec \rho_{\text{label}} \), so \( r_{\text{stamp}} \prec l \) as well. The hypotheses imply \( \text{dom}(m_1) = \text{dom}(m_2) \) and \( \text{dom}(m_1') = \text{dom}(m_2') \), and therefore 
\[
\begin{align*}
(T, r) \in \text{dom}(m_1) &\iff (T, r) \in \text{dom}(m_2) \\
(T, r) \in \text{dom}(m_1') &\iff (T, r) \in \text{dom}(m_2').
\end{align*}
\]

Since \( (T, r) \in \text{dom}(m''_i) \) and \( (T, r) \in \text{dom}(m''_j) \), there are two cases to consider.

• \( (T, r) \in \text{dom}(m_i') \) and \( (T, r) \in \text{dom}(m_j') \). In this case, for every \( i \in \{1, 2\} \) we have \( m''_i(T, r) = m_i'(T, r) \), and we conclude by using the assumption \( m_i' =_1 m_i' \).

• \( (T, r) \in \text{dom}(m_i) \) and \( (T, r) \in \text{dom}(m_j) \) while \( (T, r) \not\in \text{dom}(m_i') \) and \( (T, r) \not\in \text{dom}(m_j') \). In this case, for every \( i \in \{1, 2\} \) we have \( m''_i(T, r) = m_i(T, r) \), and we conclude by using the assumption \( m_i' =_1 m_i' \).

**Lemma A.6.** We have

1. return \(_1\) return : \( X \rightarrow \text{LIO}_{i,j}(X) \)
2. bind \(_1\) bind : \( \text{LIO}_{l_1,i,j}(X) \times (X \rightarrow \text{LIO}_{l_2,j,i}(Y)) \rightarrow \text{LIO}_{l_1,i,j}(Y) \)

**Proof.** For the first point, unfolding definitions, we have to show that whenever \( x_1 =_1 x_2, m_1 =_1 m_2 \) and \( l' \in \downset l \), we have \( m_1 =_1 m_2 \) (which was assumed) and \( x_1 \uplus l' =_1 x_2 \uplus l' \) (which follows from Lemma A.4).

For the second point, suppose that we have \( f_1 =_1 f_2 : \text{LIO}_{l_1,i,j}(X) \) and \( g_1 =_1 g_2 : X \rightarrow \text{LIO}_{l_2,j,i}(Y) \). We must show that \( \text{bind}(f_1, g_1) =_1 \text{bind}(f_2, g_2) : \text{LIO}_{l_1,i,j}(Y) \). Unfolding what this means, suppose that 
\[
\text{bind}(f_1, g_1)(m, l_i) = (m''_i, v_i', l''_i) \quad (i \in \{1, 2\}),
\]
with \( m_1 =_1 m_2 \) and \( l_i \in \downset l_i \). We have to show that \( m''_1 =_1 m''_2 \) and \( v_i' @ l''_i =_1 v_i' @ l''_i \). By the definition of bind, there are \( m_i'', v_i', l''_i \), and \( l''_i \) for \( i \in \{1, 2\} \) such that 
\[
f_i(m, l_i) = (m''_i, v_i', l''_i) \quad g_i(v_i')(m''_i, l''_i) = (m_i'', v_i', l''_i)
\]

The hypothesis on \( f_i \) implies \( m_i'' = m_i'' \) and \( v_i' @ l''_i =_1 v_i' @ l''_i \). By Lemma A.4, there are two cases to consider. If \( l''_i = l''_i \prec l, \) we know that \( v_i' =_1 v_i' \). By the hypothesis on the \( g_i \), we find that \( g_i(v_i') =_1 g_i(v_i') \), implying \( m''_i = m''_i \) and \( v_i' @ l''_i =_1 v_i' @ l''_i \) and concluding this case. Otherwise, \( l''_i \prec l \), and we conclude with Lemma A.2.

We now show that the primitives used to define the semantics also preserve indistinguishability.

**Lemma A.7** (Label cast noninterference). If \( i_1 \prec i_2 \), we have 
\[
\begin{align*}
\llbracket i_1 \prec i_2 \rrbracket &= \llbracket i_1 \prec i_2 \rrbracket \quad \text{LIO}_{l_1,i,j}(1) \quad (\text{cast noninterference})
\end{align*}
\]

**Proof.** Suppose that we have \( m_1 =_1 m_2, l', m_1, m'_2, l_1 \) and \( l_2 \) such that 
\[
\begin{align*}
\llbracket i_1 \prec i_2 \rrbracket (m_1, l') &= (m_1, 1, l_1) \\
\llbracket i_1 \prec i_2 \rrbracket (m_2, l') &= (m_2, 1, l_2)
\end{align*}
\]

We need to show that \( m_1 \uplus m'_2 = m_2 \uplus m'_2 \) and \( 1 \uplus l_1 =_1 1 \uplus l_2 \).

The last point follows by Lemma A.4, since \( 1 =_1 1 \) by definition. As for the first point, the definition of \( \llbracket i_1 \prec i_2 \rrbracket \) implies \( m_1 = m_2 = \varnothing, l_1 = l_2 = l' \) and \( l' \in \downset l \), and we conclude with the assumption \( m_1 =_1 m_2 \).

**Lemma A.8** (Cast noninterference). If \( T \prec S \), then \( \llbracket T \prec S \rrbracket = \llbracket T \prec S \rrbracket \) for all \( l \).

**Proof.** We must show that applying a cast to indistinguishable values yields indistinguishable results. We proceed by induction on the derivation of \( T \prec S \). The cases of Unit, Bool, Label and function types follow by composition, Lemma A.7, and the induction hypotheses.

For \( f = [\text{Lab}_{l_1}(T_1) \times \text{Lab}_{l_2}(T_2)], \) let \( v_1 @ l_1 =_1 v_2 @ l_1 \) be indistinguishable values. Assume that \( (v_1 @ l_1) \) and \( f(v_1 @ l_1) \) succeed, otherwise the result is trivial. It must be the case that there are values \( v_1 [T_1 \prec T_2] (v_1) \) and \( v_2 [T_1 \prec T_2] (v_2) \), so that \( f(v_1 @ l_1) = \text{return}(v_2 @ l_1) \) for all \( i \in \{1, 2\} \). We conclude by combining Lemma A.6 with the induction hypotheses on \( T_1 \) and \( T_2 \).

**Lemma A.9**. Each primitive \( f : X \) in Figure 14 satisfies \( f =_1 \) : \( X \).

**Proof.** Case unlabel. Suppose we are given labeled values \( v_1 @ l' =_1 v_2 @ l' \), memories \( m_1 =_1 m_2 \) and a label \( l_1 \). Unfolding definitions, we have to show that \( m_1 =_1 m_2 \) (obvious) and \( v_1 @ (l' \uplus l_1) =_1 v_2 @ (l' \uplus l_1) \) (which follows from Lemma A.4).

Case get\(_{l_1,b,T} \). Suppose we are given a reference \( r \), memories \( m_1 =_1 m_2 \), and a PC label \( l_1 \). We can assume that
\( m(T, r) = v_l \) for some \( v_l \), otherwise get returns an error and the result is trivial. We have to show that \( v_l \oplus (l_1 \oplus \text{label}) \equiv_l v_l \oplus (l_2 \oplus \text{label}) \). By Lemma A.4, it suffices to show that \( v_l \oplus \text{label} \equiv_l v_l \oplus \text{label} \), which follows from the hypothesis on the memories.

Case set \( i, l, T \). Suppose we are given a reference \( r \), values \( v_1 \equiv_l v_2 \), memories \( m_1 \equiv_l m_2 \) and a PC label \( l_1 \). We can assume that \( l_1 \equiv \text{label} \) and \( (T, r) \in \text{dom}(m_l) \) for all \( i \), otherwise set returns an error and the result is trivial. We have to show \( 1@l_1 \equiv_l 1@l_1 \), which is trivial, and
\[
m_1 \triangleright (T', r \mapsto v_1) \equiv_l m_2 \triangleright (T', r \mapsto v_2),
\]
which follows from Lemma A.5.

Note that the side conditions \( l_1 \equiv \text{label} \) and \( (T, r) \in \text{dom}(m_l) \) are not invoked to show noninterference for this case. Instead, they are needed to ensure that set returns an element of LIO, whose properties guarantee that bind respects indistinguishability.

Case new \( i, l, T \). Let \( v_1 \equiv_l v_2 \) be values, \( m_1 \equiv_l m_2 \) be memories, \( l_1 \) be a PC label, and \( l_2 \) be a label for the new reference. Assume that \( l_1 \equiv l_2 \), otherwise the two sides raise an error and the result is trivial. The hypotheses on \( m_1 \) and \( m_2 \) imply that \( \min(n \in \text{dom}(m_l)) \) is the same for \( i = 1 \) and \( i = 2 \). Call this number \( n \), and set \( r = (n, l_1, l_2) \). The results of each execution are of the form \( (T', r \mapsto v_1), r, l_1 \) and \( (T', r \mapsto v_2), r, l_2 \), and we have to show that \( m_1 \triangleright (T', r \mapsto v_1) \equiv_l m_2 \triangleright (T', r \mapsto v_2) \) and \( r@l_1 \equiv r@l_1 \). The first point follows by Lemma A.5. As for the second point, it holds by Lemma A.4, since \( r \equiv_l r \) holds trivially.

Case toLab \( i, l, T \). Suppose that we are given \( l_1 \in \gamma(l_1), f_1 \equiv_l f_2 : \text{LIO}_{i,l_1}(X), m_1 \equiv_l m_2 \). By the definition of toLab, we can suppose that \( f(x, l_1) = (m'_f, v_i, l'_l) \) and \( l'_l \equiv l_1 \lor l_2 \) for some \( m'_f, v_i \) and \( l'_l \), with \( i \in \{1, 2\} \). We have to show that \( m_1 \triangleright m'_f \equiv_l m_2 \triangleright m'_f \) and \( v_i@l_1 \triangleright l_2 \equiv_l v_i@l_2 \triangleright l_2 \). By the hypothesis on the \( f \), we know that the first condition holds, and also that \( v_i@l_i \equiv_l v_i@l_i \) if \( l_i \equiv l \). We know that \( l_i \) and \( l'_i \) are below \( I \). As above, \( v_i \equiv_l v_2 \), and we conclude by applying Lemma A.4.

Taken together, these lemmas lead to our main result.

**Theorem 5.1** (Noninterference). If \( \Gamma \vdash_{i,l_1} e : T \), we have
\[
\mathbb{C}[e] \equiv_l \mathbb{C}[e] : \Gamma^\text{cont} \text{LIO}_{i,l_1}(\Gamma[T]).
\]

*Proof.* By induction on the typing derivation of \( e \), combining the previous preservation results. We detail some cases here.

Case pcLabel. Suppose we are given memories \( m_1 \equiv_l m_2 \) and a PC label \( l_1 \). By Lemmas A.4 and A.5, we know \( \emptyset \equiv_l \emptyset \) and \( l_1 \leftarrow l_1 \equiv_l l_1 \leftarrow l_1 \), which concludes this case.

Case fun. It suffices to show that \( \lambda v.\mathbb{C}[e](s_1[x \mapsto v]) \equiv_l \lambda v.\mathbb{C}[e](s_2[x \mapsto v]) \) assuming that \( \mathbb{C} \equiv_l \mathbb{C} \) and \( s_1 \equiv_l s_2 \).

Thus, we have to show \( \mathbb{C}[s_1[x \mapsto v]] = \mathbb{C}[s_2[x \mapsto v]] \) for all arguments \( v_1 \equiv_v v_2 \). This follows because \( s_1[x \mapsto v_1] \equiv_s s_1[x \mapsto v_2] \), and by the definition of indistinguishability for functions.

Case app. By composition, it suffices to show that the last clause of the semantics of app respects indistinguishability. Unfolding what this means, we have to show that \( s_1(f)(v_1) \equiv_l s_2(f)(v_2) \) whenever \( s_1 \equiv_l s_2 : \Gamma[f] = T \Rightarrow S \) and \( v_1 \equiv_l v_2 : [T] \). This follows by the definition of indistinguishability for \( [T \Rightarrow S] \).

**A.4 Proving the gradual guarantees**

The construction of the error approximation relations is similar to indistinguishability, but takes place in the admissible CPO\_\text{Type} fibrations \( r : \text{Appr} \Rightarrow \text{CPO}_{\text{Type}} \) defined as follows. An object of \text{Appr} is a pair \((X, R)\) where \( X \in \text{CPO}_{\text{Type}} \) and \((\text{R}_T \triangleleft \text{S}) \in X(\text{T} \times X(\text{S})) \) is a family of chain-complete relations. A morphism of type \((X, R) \Rightarrow (Y, U)\) is a family of partial functions \((f_T : X(\text{T}) \Rightarrow Y(\text{S}))_{\text{Type}} \) such that, for every \((x_T, x_S) \in R(\text{T} \triangleleft \text{S}), f_T(x_T) = f_S(x_S) = \perp\) or \( f_T(x_T) = y_T \in Y(\text{T}), f_S(x_S) = y_S \in Y(\text{S}) \) and \((y_T, y_S) \in U(\text{T} \triangleleft \text{S})\).

We lift the function \( f^\Rightarrow \) on \text{Appr}.
\[
\text{Appr} \text{op} \times \text{Appr} \xrightarrow{f^\Rightarrow} \text{Appr}
\]

whose definition is given in Figure 19, following similar conventions as in Appendix A.3. This allows us to construct \( (\text{R}^*_T(\text{T} \triangleleft \text{S}) \subseteq D(T) \times D(S)) \) such that
\[
(f_T(x_T), f_S(x_S)) \in R^*_T(\text{T} \triangleleft \text{S})
\]

which we take as the definition of the error approximation relations of Figure 19.

Having defined error approximation, we are ready to prove the gradual guarantees. We begin with a few auxiliary results that show that loosening the labels in a program does not interfere with subtyping or joins.

**Lemma A.10** (Dynamism and subtyping). If \( l_1 \ll l_2 \), then \( l_1 \ll l_2 \) implies \( l_2 \ll l_1 \) and \( l_1 \ll l_1 \) implies \( l_1 \ll l_2 \). If \( T_1 \ll T_2 \), then \( T_1 \ll S \) implies \( T_2 \ll S \) and \( S \ll T_1 \) implies \( S \ll T_2 \). In particular, since all relations are reflexive, \( \ll \) and \( \ll \) are contained in \( \subseteq \).

In light of this lemma, we will write \([T \ll S]\) (or \([T \ll S]\)) instead of \([T \subseteq S]\) when \( T \ll S \) (or \( T \ll S \)) holds. We will adopt a similar convention for label casts.
\[ \hat{F}_R^2(R^-, R^+)(T = T) \triangleq \{ (x, x) \in X(T) \times X(T) \} \]
\[ (T = \text{Unit, Bool, Label}) \]
\[ \hat{F}_R^2(R^-, R^+)(\text{Ref}(T) \iff \text{Ref}_2(T')) \triangleq \{ (x, x) \mid x \in X(\text{Ref}(T)) \} \]
\[ \hat{F}_R^2(R^-, R^+)(\text{Lab}(T) \iff \text{Lab}_2(T')) \triangleq \{ (x@l, x'@l) \mid (x, x') \in R^+(T \prec T') \} \]
\[ \hat{F}_R^2(R^-, R^+)(T_1 \xrightarrow{l_1, l_2} T_2 \prec T_1' \xrightarrow{l_1', l_2'} T_2') \triangleq \{ (f, f') \mid \forall (x_1, x_2) \in R^+(T_1 \prec T_1'), (m_1, m_1') \in G^*_\text{Mem}(R), l_1, l_1' \}
\[ (f(x_1)(m_1, l_1) = y \iff f'(x_1)(m_1', l_1') = y) \]
\[ \land \forall x_2, m_2, l_2, f(x_2)(m_2, l_2) = \exists x_2', m_2', f'(x_2')(m_2', l_2') \in R^+(T_2 \prec T_2') \land (m_2, m_2') \in G^*_\text{Mem}(R') \} \]
\[ G^*_\text{Mem}(R) \triangleq \{ (m_1, m_2) \mid \text{dom}(m_1) = \text{dom}(m_2) \}
\[ \land \forall (T, r) \in \text{dom}(m_1), (m_1(T, r), m_2(T, r)) \in R(T \prec T) \} \]

**Figure 24.** Lifting of the functor \( F \) for error approximation. The clause for \( \text{Ref} \) assumes that \( X(\text{Ref}(T)) \subseteq X(\text{Ref}_2(T')) \) when \( l \prec l' \). Strictly speaking, this is not valid for the entire domain of definition of the functor \( F \), but we can show that \( F \) does preserve this property, so there is no harm in assuming it holds.

**Lemma A.11 (Dynamism, joins and meets).** Let \( \circ \in \{ \lor, \land \} \).
\[ \text{If } l_1 = l'_1 \text{ and } l_2 = l'_2 \text{ then } l_1 \lor l_2 = l'_1 \lor l'_2. \]
\[ \text{If } T = T', S = S', \text{ and } T \subseteq S \text{ is well-defined, then so is } T' \subseteq S'. \]

**Lemma A.12.** If \( l \prec l' \), then \( y(l) \subseteq y(l') \) and \( y(l) \subseteq y(l') \).

**Theorem 6.2 (Static Gradual Guarantee).** If \( T \prec T', l_1 \prec l'_1, e \prec e' \), and \( T \xrightarrow{l_1, l_2} e : T \), there exist \( l_2' \succ l_2 \) and \( T' = T \) such that \( \Gamma' \mapsto l_2' e' : T' \).

**Proof.** By induction on the typing derivation, using Lemmas A.10 to A.12. \( \square \)

Next, we cover a few properties of the error approximation relation.

**Lemma A.13.**

1. \( \emptyset \lor \emptyset \)
2. If \( m_1 = m'_1 \) and \( m_2 = m'_2 \), then \( m_1 \lor m_2 = m'_1 \lor m'_2 \).
3. If \( T \prec S \) and \( x = y : T \prec S \), then \( [T, r \mapsto x] = [S, r \mapsto y] \).

**Lemma A.14.** If \( l_i \prec l'_i \) for \( i \in \{ 1, 2, 3 \} \), \( T = T' \) and \( S = S' \), then

1. \( \text{return}_{l_1, l_2, l_3}[T] = \text{return}_{l_1', l_2', l_3'}[T'] \)
2. \( \text{bind}_{l_1, l_2, l_3}[T][S] = \text{bind}_{l_1', l_2', l_3'}[T'][S'] \)

**Proof.** For return, suppose that we have values \( v \lor v' \), memories \( m = m' \) and a label \( l \). We must show that \( \emptyset \lor \emptyset \) (trivial) and \( v \lor v' \) (which follows from the assumption).

For bind, suppose that we have computations
\[ f_1 = f_2 \iff \text{LIO}_{l_i, l'_i}[T] = \text{LIO}_{l_i, l'_i}[T'] \],
functions
\[ g_1 = g_2 \iff \text{LIO}_{l_i, l'_i}[S] = \text{LIO}_{l_i, l'_i}[S'] \],
memories \( m_1 = m_2 \), and a label \( l \). We must show that
\[ \text{bind}(f_1, g_1)(m_1, l) = \text{bind}(f_2, g_2)(m_2, l) \].

First, suppose that \( \text{bind}(f_1, g_1)(m_1, l) = \emptyset \). There are two cases to consider. If \( f_1(m_1, l) = \emptyset \), we have \( f_2(m_2, l) = \emptyset \) by the assumption on the \( f_i \), and thus \( \text{bind}(f_2, g_2)(m_2, l) \) also diverges. Otherwise, it must be the case that \( m_1 = m'_1 \), \( x_1 \) and \( l' \), such that \( f_1(m_1, l) = (m'_1, x_1, l') \) and \( g_1(x_1)(m_1 \lor m_1') = \emptyset \). By the hypothesis on the \( f_i \), we find \( m_2' \) and \( x_2 \), such that \( f_2(m_2, l) = (m_2', x_2, l') \), \( m_1' = m_2' \) and \( x_1 = x_2 \).

By Lemma A.13, we find that \( m_1 \lor m_2 = m_1' \lor m_2' = m_1' \). By the hypothesis on \( g_i \), we find that \( g_2(x_2)(m_2 \lor m_2', l') = \emptyset \), which concludes this case.

Now, suppose that \( \text{bind}(f_1, g_1)(m_1, l) = (m', x', l') \). By the definition of bind, we find \( m_1 = m_2 = m_1' = m_2' \), \( x_1 = x_1' \) and \( x_2 = x_2' \) such that
\[ f_1(m_1, l) = (m'_1, x_1, l'), \quad g_1(x_1)(m_1 \lor m_1') = (m_1', x_1', l'') \]
\[ m_1'' = m_1' \lor m_1'' \]

Once again, the hypothesis on the \( f_i \) and \( g_i \) combined with Lemma A.13 allow us to find \( m_2' = m_1' \), \( x_2 = x_1 \), \( m_2'' = m_1'' \) and \( x_2' = x_1' \) such that
\[ f_2(m_2, l) = (m_2', x_2, l') \quad g_2(x_2)(m_2 \lor m_2', l') = (m_2'', x_2', l'') \]

which allows us to conclude. \( \square \)

**Lemma A.15.** Suppose we have gradual labels \( l_1, l_1', l_2 \) and \( l_2' \) such that \( l_1 = l'_1 \) for all \( i \in \{ 1, 2 \} \) and \( l_1 \prec l_2 \) (and thus \( l_1' \prec l_2' \) by Lemma A.10). Then \( [l_1 \lor l_2] \subseteq [l_1' \lor l_2'] \).

**Proof.** Suppose that we are given a label \( l_1 \in [l_1 \lor l_2] \) and two memories \( m = m' \). If \( l_1 \not\subseteq l_2 \), we have \( [l_1 \prec l_2](m, l_1) \) is error, and the result is trivial. Otherwise, we find \( l_1 \subseteq l_2 \) and we can conclude because both executions result in \( (\emptyset, 1, l_1) \). \( \square \)
Lemma A.16. Suppose that we have gradual types \( T_1, T'_1, T_2 \) and \( T'_2 \) such that \( T_1 \prec T'_1 \) for all \( i \in \{1, 2\} \) and \( T_1 \prec T_2 \) (and thus \( T'_1 \prec T'_2 \)) by Lemma A.10. Then \( \llbracket T_1 \ltimes T_2 \rrbracket = \llbracket T'_1 \ltimes T'_2 \rrbracket \).

Proof. By unfolding definitions, we must show that related values are mapped to related results. We proceed by induction on \( T_1 \) and inversion on the derivations relating the types. The cases of Unit, Bool and Label are trivial, since they reduce to return the result. The case of function types follows by composition, the induction hypotheses, and by applying Lemma A.15.

It remains to show the result for \( T_1 = \text{Ref}_i(S_1) \) and \( T_1 = \text{Lab}_i(S_1) \). We focus on the second case, since the first one is similar. The remaining types are of the form

\[
T'_1 = \text{Lab}_i(S'_1) \quad T_2 = \text{Lab}_j(S_2) \quad T'_2 = \text{Lab}_j(S'_2),
\]

with \( S_1, S_2, S'_1, S'_2, I_1 \prec I'_1 \) and \( I_2 \prec I'_2 \). Suppose that we are given related labeled values \( v_1@I_1 \prec v'_1@I'_1 : \text{Lab}_i(S'_1) \llbracket S_1 \rrbracket \llbracket S'_1 \rrbracket \). If \( I_1 \notin I'_1 \), we have \( \llbracket T_1 \ltimes T_2 \rrbracket (v_1@I_1) = \lambda(-) \text{ error} \), and the result is trivial. Otherwise, we have \( I_1 \in I'_1 \), along \( I'_2 \), which implies

\[
\llbracket T_1 \ltimes T_2 \rrbracket (v_1@I_1) = \begin{cases} v_2 \leftarrow \llbracket S_1 \ltimes S_2 \rrbracket (v_1); \text{return}(v_2) @I'_1 \end{cases}
\]

\[
\llbracket T'_1 \ltimes T'_2 \rrbracket (v'_1@I'_1) = \begin{cases} v'_2 \leftarrow \llbracket S'_1 \ltimes S'_2 \rrbracket (v'_1); \text{return}(v'_2) @I'_1 \end{cases}
\]

We conclude by composition, applying the induction hypotheses on \( S_1, S_2, S'_1 \) and \( S'_2 \).

\[ \square \]

Corollary A.17. If \( T_1 \prec T'_1 \) and \( T_2 \prec T'_2 \), then \( \llbracket T_1 \ltimes T_2 \rrbracket = \llbracket T'_1 \ltimes T'_2 \rrbracket \).

Proof: If \( T_1 \prec T_2 \), then \( T'_1 \prec T'_2 \) by Lemma A.10, and the result is equivalent to \( \llbracket T_1 \ltimes T_2 \rrbracket = \llbracket T'_1 \ltimes T'_2 \rrbracket \), which follows from Lemma A.16. Otherwise, \( \llbracket T_1 \ltimes T_2 \rrbracket = \lambda(-) \text{ error} \), and the result is trivial.

\[ \square \]

Lemma A.18. Each primitive \( f \) of Figure 14 satisfies \( f \prec f \) (given suitable parameters).

Proof. By “given suitable parameters,” we mean that each primitive is parameterized by CPOs and labels, and we must choose the parameters correctly for each side of the above relation for it to hold. This means that the labels on the left-hand side must be less dynamic than those on the right-hand side. We provide more precise statements for each case below.

Case unlabeled. We want to show \( \text{unlabel}_i(I_1, I_2) \prec \text{unlabel}_i(I'_1, I'_2) \) when \( I_1 \prec I'_1 \) for all \( i \). Suppose that we have a label \( I_1 \subseteq I'_1 \subseteq I'_2 \), labeled values \( v @ I_2 \prec v' @ I'_2 \) and memories \( m_1 \prec m'_1 \).

By definition, we have \( I_2 = I'_2 \) and \( v \prec v' \), and the results of executing unlabeled are \( (\varnothing, v, I_1 \prec I_2) \) and \( (\varnothing, v', I_1 \prec I_2) \). Thus, we have to show \( \varnothing \prec \varnothing \) (trivial) and \( v \prec v' \), which follows from the assumption.

Case get. Given \( I_1 \prec I'_1 \) for all \( i \in \{1, 2\} \) and \( T \prec T' \) we must show

\[
\text{get}_{I_1, I'_1, T} \prec \text{get}_{I'_1, I'_1, T'},
\]

\[
: \text{Ref}_{I_1} \xrightarrow{\text{cont}} \text{LIO}_{I_1, I'_1, T} \prec \text{Ref}_{I'_1} \xrightarrow{\text{cont}} \text{LIO}_{I'_1, I'_1, T'}.
\]

(Recall that \( T' \) = \( T' \); cf. Lemma A.1.)

Suppose that we are given a label \( I_1 \subseteq I'_1 \subseteq I'_2 \), along elements

\[
r \prec r : \text{Ref}_{I_1} \prec \text{Ref}_{I'_1} \quad m_1 \prec m'_1 : \text{Mem} \prec \text{Mem}.
\]

If \( (T, r) \not\in \text{dom}(m_1) = \text{dom}(m_2) \), both executions return error and the result is trivial. Otherwise, \( (T, r) \in \text{dom}(m_1) = \text{dom}(m_2) \), and the executions return

\[
(\varnothing, m_1(T', r), I_1 \vdash m'_1) \text{ and } (\varnothing, m'_1(T', r), I_1 \vdash m_1).
\]

We conclude using \( m_1(T', r) \prec m'_1(T', r) \), a consequence of \( m_1 \prec m_2 \).

Case set. Given \( I_1 \prec I'_1 \) for all \( i \in \{1, 2\} \) and \( T \prec T' \), we have to show that

\[
\text{set}_{I_1, I'_1, T} \prec \text{set}_{I'_1, I'_1, T'},
\]

\[
: \text{Ref}_{I_1} \xrightarrow{\text{cont}} \text{LIO}_{I_1, I'_1, T} \prec \text{Ref}_{I'_1} \xrightarrow{\text{cont}} \text{LIO}_{I'_1, I'_1, T'}.
\]

(As in the get case, \( T \) does not have to vary.)

Suppose we are given a label \( I_1 \subseteq I'_1 \subseteq I'_2 \) along elements

\[
r \prec r : \text{Ref}_{I_1} \prec \text{Ref}_{I'_1} \quad v \prec v' : [T] = [T']
\]

\[
m_1 \prec m'_1 : \text{Mem} \prec \text{Mem}.
\]

Suppose that \( m(T', r) \in \text{dom}(m_1) = \text{dom}(m'_1) \) and \( I_1 \vdash m'_1 \); otherwise the result follows because

\[
\text{set}(r, v)(m_1, I_1) = \text{set}(r, v')(m'_1, I_1) = \text{error}.
\]

By unfolding the definition of set, we must show that \( T' \prec T \prec v | \prec [T'] \prec [T'] \) (follows by Lemma A.13) and \( 1 = 1 \) (trivial).

Case new. We must show that \( \text{new}_{I_1, I'_1, T} \prec \text{new}_{I'_1, I'_1, T'} \) at the CPOs

\[
\gamma(I_1) \xrightarrow{\text{cont}} \text{LIO}_{I_1, I'_1, T} \prec \gamma(I'_1) \xrightarrow{\text{cont}} \text{LIO}_{I'_1, I'_1, T'}.
\]

when \( I_1 \prec I'_1 \) for every \( i \) and \( T \prec T' \). (As in the get case, \( T \) does not need to vary.)

Suppose we are given \( v \prec v' : [T] = [T'] \), \( I_1 \subseteq I'_1 \subseteq I'_2 \) and memories \( m_1 \prec m'_1 \).

We have to show that \( \text{new}(I_1, v)(m_1, I_2) \prec \text{new}(I'_1, v')(m'_1, I'_2) \).

Suppose that \( I_2 \prec I_1 \), otherwise both terms are equal to \( \text{error} \) and the result is trivial. In this case, since \( \text{dom}(m_1) = \text{dom}(m'_1) \), the results are of the form \( ([T], r \mapsto v, r, I_2) \) and \( ([T'], r \mapsto v', r, I'_2) \) for some reference \( r \), and we can conclude.
We proceed by induction on the derivation of \( \overline{l} \) whenever \( \overline{i} \). Suppose that we have labels \( l_1 \in \gamma(h_1) \subset \gamma(h'_1) \), \( l_2 \in l'_2 \subset l'_2 \) and
\[
f \to f' : \text{LIO}_{l_1, l_2}(X) \to \text{LIO}_{l'_1, l'_2}(X')
\]
\[m_1 \sim m'_1 : \text{Mem} \sim \text{Mem}.
\]

There are the following cases to consider.

1. If \( \text{toLab}(l_1, f)(m_1, l_2) = \bot \), we know that \( f(m_1, l_2) = \bot \).
   This implies \( \text{toLab}(l_1, f')(m'_1, l_2) = \bot \) by the above hypothesis, and the conclusion follows.

2. If \( \text{toLab}(l_1, f)(m_1, l_2) = \text{error} \), the relation reduces to \( \text{error} \sim \text{toLab}(l_1, f')(m'_1, l_2) \), which also holds trivially.

3. Otherwise, \( \text{toLab}(l_1, f)(m_1, l_2) \) must be of the form \( (m_2, v_2) @ l_1, l_2 \), with \( f(m_1, l_2) = (m_2, v_2) \) and \( l_1 \sim l'_1 v_2 \). Since \( f \to f' \), it must be the case that \( f(m'_1, l_2) = (m'_2, v'_2) \) with \( m_2 \sim m'_2 \) and \( v_2 \sim v'_2 \). We conclude that \( \text{toLab}(l_1, f')(m'_1, l_2) = (m'_2, v'_2) @ l_1, l_2 \), which implies the final result.
\[\square\]

**Lemma A.19 (Unique Typing).** If \( \Gamma \vdash \overline{t}, \overline{i}, t : T \) and \( \Gamma \vdash \overline{t}, \overline{i}, t' : T' \), then \( \overline{l}_2 = \overline{l}'_2 \) and \( T = T' \).

**Proof.** By induction on \( e \) and inversion on the typing derivations.
\[\square\]

**Theorem 6.3 (Dynamic Gradual Guarantee, General).** If \( \Gamma \vdash \overline{t}, \overline{i}, e : T \), \( \Gamma' \vdash \overline{t}, \overline{i}, e' : T' \), \( \Gamma = \Gamma' \), \( \overline{l}_i = \overline{l}'_i \) (\( i \in \{1, 2\} \)), \( e \sim e' \) and \( T \sim T' \), then \( [e] = [e'] : [\Gamma] \xrightarrow{\text{cont}} \text{LIO}_{l_1, l_2}([T]) \sim [\Gamma'] \xrightarrow{\text{cont}} \text{LIO}_{l'_1, l'_2}([T']) \).

**Proof.** By Theorem 6.2 and Lemma A.19, the typing derivation of \( e' \) is entirely determined by that of \( e \), plus \( \Gamma' \) and \( \overline{l}_1 \).
We proceed by induction on the derivation of \( e' \) and inversion on the derivation of \( e' \) for each case. Most cases follow by composition, induction hypotheses, and the results of Lemma A.18; we detail the interesting ones here.

**Case pcLabel.** Trivial: the results depend only on the PC label, which is the same on both sides.

**Case fun.** We have \( e = \text{fun}(x : T, S, e_1) \) and \( e' = \text{fun}(x : T, S', e_1') \) with \( \overline{l} \sim \overline{l}' \), \( S \sim S' \) and \( e_1 \sim e_1' \). By the induction hypotheses, we know that \( [e_1] \sim [e_1'] \), implying that \( [e_1](s[x \mapsto v]) \sim [e_1'](s'[x \mapsto v']) \) for all environments \( s \sim s' \) and arguments \( v \sim v' \). By definition, \( \lambda x. [e_1](s[x \mapsto v]) \sim \lambda x. [e_1'](s'[x \mapsto v']) \), hence \( [e](s) \sim [e'](s') \), concluding this case.

**Case app.** We have \( e = e' = \text{app}(f, x) \), with
\[
\begin{align*}
\Gamma(f) : S_1 & \xrightarrow{i} T \\
\Gamma(x) : S_2 & \\
\Gamma'(f) : S'_1 & \xrightarrow{i} T' \\
\Gamma(x) : S'_2,
\end{align*}
\]
such that \( \overline{l}_3 \sim \overline{l}_3' \), \( S_1 \sim S'_1 \), \( S_2 \sim S'_2 \) and \( S_1 \sim S'_1 \). Given environments \( s \sim s' \), after performing the appropriate checks, it suffices to show that \( s(f)(v) \sim s'(f)(v') \) for \( v \sim v' \).
This follows from the definition of error approximation for environments and for functions.
\[\square\]