Types for Information Flow Control: Labeling Granularity and Semantic Models

Vineet Rajani
Max Planck Institute for Software Systems
Saarland Informatics Campus
Germany

Deepak Garg
Max Planck Institute for Software Systems
Saarland Informatics Campus
Germany

***Abstract*—Language-based information flow control (IFC) tracks dependencies within a program using sensitivity labels and prohibits public outputs from depending on secret inputs. In particular, literature has proposed several type systems for tracking these dependencies. On one extreme, there are fine-grained type systems (like Flow Caml) that label all values individually and track dependence at the level of individual values. On the other extreme are coarse-grained type systems (like HLIO) that track dependence coarsely, by associating a single label with an entire computation context and not labeling all values individually.**

In this paper, we show that, despite their glaring differences, both these styles are, in fact, equally expressive. To do this, we show a semantics- and type-preserving translation from a coarse-grained type system to a fine-grained one and vice-versa. The forward translation isn't surprising, but the backward translation is: It requires a construct to arbitrarily limit the scope of a context label in the coarse-grained type system (e.g., HLIO's "toLabeled" construct). As a separate contribution, we show how to extend work on logical relation models of IFC types to higher-order state. We build such logical relations for both the fine-grained type system and the coarse-grained type system. We use these relations to prove the two type systems and our translations between them sound.

1. Introduction

Information flow control (IFC) is the problem of tracking flows of information within a computer system and controlling or prohibiting flows that contravene the policy in effect. In a language-based setting, IFC requires tracking *dependencies* between a program's inputs, intermediate values and outputs. This can be done dynamically with runtime monitoring [1], [2] or statically using some form of abstraction interpretation such as a type system. Our focus in this paper is the second of these methods—IFC enforced through type systems. In fact, literature has proposed several type systems for IFC, e.g., [3], [4], [5], [6], [7], [8], [9], [10], [11]. All these type systems have one aspect in common: They all introduce *security labels* or *levels*, elements of a security lattice, that abstract program values. These labels are used to track dependencies between program values.

A significant design consideration for an IFC type system is the granularity (or extent) of the label abstraction, and the effect of this granularity on the expressiveness of the type system. By expressiveness here we mean the ability of a type system to type as many semantically secure programs as possible.[1] More specifically, we call a type system T more expressive than a type system T' if there is a compositional, semantics-preserving transformation of programs typed under T' to programs typed under T.[2]

The question of granularity has at least two aspects. First, one may vary the granularity of the *labels* themselves. For example, a fine-grained label on a value may precisely specify the program variables or inputs that the value depends on. On the other hand, a coarse-grained label may specify only an upper-bound on the confidentiality level (e.g., "secret", "top-secret", etc.) of all inputs on which the labeled value depends. The effect of varying this notion of granularity (of the labels) on the expressiveness of the type system has been studied in prior work [14].

A different kind of granularity, whose expressiveness is the focus of this paper, is the granularity of *labeling* (not *labels*).[3] Here, a *fine-grained type system* is one that labels every program value individually. For instance, Flow Caml, a type system for IFC on ML [3], adds a label on every type constructor and, hence, on every value, top-level and nested. As an example, the type $(A^H \times B^L)^L$ might ascribe low (public) pairs, whose first projection is high (private) and whose second projection is low. (H and L are standard labels for high and low confidentiality data, respectively.) Since fine-grained type systems label individual values, they also track dependencies at the granularity of individual values. For example, combining a high value with a low value using a primitive operator in the language results in a high value. Many other type systems are similarly fine-grained [4], [5], [6], [7].

In contrast, a *coarse-grained type system* labels an entire sub-computation using a single label. All values produced

1. No sound type system can type all semantically secure programs, since freedom from bad flows (specifically, the standard information flow security property called noninterference [12]) is undecidable.

2. This notion of expressiveness is closely related to what Felleisen calls macro expressibility [13].

3. In the rest of the paper, granularity refers to the granularity of labeling, not the granularity of labels.

within the scope of the sub-computation implicitly have that label. Hence, it not necessary to label individual values. As an example, the SLIO and HLIO systems [10] introduce a monad for heap I/O (similar to Haskell's IO monad), but refine the monadic type to include two labels, ℓ_i and ℓ_o, as in $(SLIO\ \ell_i\ \ell_o\ \tau)$. This type represents stateful computations of type τ that start from the confidentiality label ℓ_i and end with the confidentiality label ℓ_o. ℓ_i is an upper-bound on the confidentiality of all prior computations that the current computation depends on; accordingly, the current computation can have write effects at levels above ℓ_i only. ℓ_o is an upper-bound on the confidentiality of the current computation; accordingly, the current computation can have read effects only at levels below ℓ_o. Importantly, there is no need to label individual values or nested types. Instead, every value produced by the current computation implicitly inherits the label ℓ_o, and labels are tracked via monadic sequencing (bind) at the granularity of computations. Other type systems [8], [9], [11] are also coarse-grained, although [8], [9] do not use monads to confine effects.

Given these vastly contrasting labeling granularities for the same end-goal—information flow control, a natural question is one of their relative expressiveness [15]. In general, it seems that fine-grained type systems should be at least as expressive as coarse-grained type systems, since the former track flows at finer granularity (individual values as opposed to entire sub-computations) and, hence, should abstract flows less than the latter. In the other direction, the situation is less clear. Upfront, it seems that coarse-grained type systems *may* be less expressive than fine-grained type systems, but then one wonders whether by structuring programs as extremely small computations in a coarse-grained type system, one may recover the expressiveness of a fine-grained type system.

In this paper, we show constructively that both these intuitions are, in fact, correct. We do this using specific instances of the two kinds of type systems in the setting of a higher-order language with state (similar to ML). For the fine-grained type system we use a system very close to SLam [7] and the exception-free fragment of Flow Caml [3]. For the coarse-grained type system we use a variant of the static fragment of HLIO [10]. This calculus has a specific construct to limit the scope of a computation's label in a safe way. We then show that well-typed programs in each type system can be translated to the other, preserving typability and meaning. This establishes that the type systems are equally expressive.

We believe this settles an open question about the relative expressiveness of setting up IFC type systems with different labeling granularities. Our result also has an immediate practical consequence: Since coarse-grained IFC type systems usually burden a programmer less with annotations (since not every value has to be labeled) and we have shown now that they are as expressive as fine-grained IFC type systems, there seems to be some merit to preferring coarse-grained IFC type systems over fine-grained ones in general.

As a second contribution of independent interest, we show how to set up semantic, logical relations models of IFC types in both the fine-grained and the coarse-grained settings, over calculi with higher-order state. While models of IFC types have been considered before [7], [11], [16], [17], we do not know of any development that covers higher-order state. In fact, models of types in the presence of higher-order state are notoriously difficult. Here, we have the added complication of information flow labels. Fortunately, enough development has occurred in the programming languages community in the past decade to give us a good starting point. Specifically, our models are based on step-indexed Kripke logical relations [18]. Like earlier work, our models are relational, i.e., they relate two runs of a program to each other. This is essential since we are interested in proving noninterference [12], the standard security property which says that public outputs of a program are not influenced by private inputs (i.e., there are no bad flows). This property is naturally defined using two runs. Using our models, we derive proofs of the soundness of both the fine-grained and the coarse-grained type systems.

We also use our logical relations to show that our translations are meaningful. Specifically, we set up *cross-language logical relations* to prove that our translations preserve program semantics, and from this, we derive a crucial result for each translation: Using the noninterference theorem of the target language as a lemma, we are able to re-prove the noninterference theorem for the source language directly. These results imply that our translations preserve label annotations meaningfully [19]. Like all logical relations models, we expect that our models can be used for other purposes as well.

To summarize, the two contributions of this work are:

- Typability- and meaning-preserving translations between a fine-grained and a coarse-grained IFC type system, showing that these type systems are equally expressive.
- Logical relations models of both type systems, covering both higher-order functions and higher-order state.

Due to lack of space, many technical details and proofs are omitted from this paper. These are provided in an appendix available online from the authors' homepages.

Note. Readers interested only in our translations but not the details of our semantic models can skip sections pertaining to the latter (e.g., Section 2.1.1). This will not affect the readability of the rest of the paper.

2. The Two Type Systems

In this section, we describe the fine-grained and coarse-grained type systems we work with. Both type systems are set up for higher-order stateful languages, but differ considerably in how they enforce IFC. The fine-grained type system, called FG, works on a language with pervasive side-effects like ML, and associates a security label with every expression in the language. The coarse-grained type system,

CG, works on a language that isolates state in a monad (like Haskell's IO monad) and tracks flows coarsely at the granularity of a monadic computation, not on pure values within a monadic computation.

Both FG and CG use security labels (denoted by ℓ) drawn from an arbitrary security lattice $(\mathcal{L}, \sqsubseteq)$. We denote the least and top element of the lattice by $\bot$ and $\top$ respectively. As usual, the goal of the type systems is to ensure that outputs labeled ℓ depend only on inputs with security labels ℓ or lower. For drawing intuitions, we find it convenient to think of a confidentiality lattice (labels higher in the lattice represent higher confidentiality). However, nothing in our technical development is specific to a confidentiality lattice—the development works for any security lattice including an integrity lattice and a product lattice for confidentiality and integrity.

2.1. The fine-grained type system, FG

FG is based on the SLam calculus [7], but uses a presentation similar to Flow Caml, an IFC type system for ML [3]. It works on a call-by-value, eager language, which is a simplification of ML. The syntax of the language is shown at the top of Figure 1. The language has all the usual expected constructs: Functions, pairs, sums, and mutable references (heap locations). The expression $!e$ dereferences the location that e evaluates to, while $e_1 := e_2$ assigns the value that e_2 evaluates to, to the location that e_1 evaluates to. The dynamic semantics of the language are defined by a "big-step" judgment $(H, e) \Downarrow_j (H', v)$, which means that starting from heap H, expression e evaluates to value v, ending with heap H'. This evaluation takes j steps. The number of steps is important only for our logical relations models. The rules for the big-step judgment are standard, hence omitted here.

Every type τ in FG, including a type nested inside another, carries a security label. The security label represents the confidentiality level of the values the type ascribes. It is also convenient to define unlabeled types, denoted A, as shown in Figure 1.

Typing rules. FG uses the typing judgment $\Gamma \vdash_{pc} e : \tau$. As usual, Γ maps free variables of e to their types. The judgment means that, given the types for free variables as in Γ, e has type τ. The annotation pc is also a label drawn from $\mathcal{L}$, often called the "program counter" label. This label is a *lower bound* on the write effects of e. The type system ensures that any reference that e writes to is at a level pc or higher. This is necessary to prevent information leaks via the heap. A similar annotation, ℓ_e, appears in the function type $\tau_1 \xrightarrow{\ell_e} \tau_2$. Here, ℓ_e is a lower bound on the write effects of the body of the function.

FG's typing rules are shown in Figure 1. We describe some of the important rules. In the rule for case analysis (FG-case), if the case analyzed expression e has label ℓ, then both the case branches are typed in a pc that is *joined* with ℓ. This ensures that the branches do not have write effects below ℓ, which is necessary for IFC since the execution of the branches is control dependent on a value (the case condition) of confidentiality ℓ. Similarly, the type of the result of the case branches, τ, must have a top-level label at least ℓ. This is indicated by the premise $\tau \searrow \ell$ and prevents implicit leaks via the result. The relation $\tau \searrow \ell$, read "τ protected at ℓ" [11], means that if $\tau = \mathsf{A}^{\ell'}$, then $\ell \sqsubseteq \ell'$.

The rule for function application (FG-app) follows similar principles. If the function expression e_1 being applied has type $(\tau_1 \xrightarrow{\ell_e} \tau_2)^{\ell}$, then ℓ must be below ℓ_e and the result τ_2 must be protected at ℓ to prevent implicit leaks arising from the identity of the function that e_1 evaluates to.

In the rule for assignment (FG-assign), if the expression e_1 being assigned has type $(\mathsf{ref}\ \tau)^{\ell}$, then τ must be protected at $pc \sqcup \ell$ to ensure that the written value (of type τ) has a label above pc and ℓ. The former enforces the meaning of the judgment's pc, while the latter protects the identity of the reference that e_1 evaluates to.

All introduction rules such as those for λs, pairs and sums produce expressions labeled $\bot$. This label can be weakened (increased) freely with the subtyping rule FGsub-label. The other subtyping rules are the expected ones, e.g., subtyping for unlabeled function types $\tau_1 \xrightarrow{\ell_e} \tau_2$ is co-variant in τ_2 and contra-variant in τ_1 and ℓ_e (contra-variance in ℓ_e is required since ℓ_e is a *lower* bound on an effect). Subtyping for $\mathsf{ref}\ \tau$ is invariant in τ, as usual.

The main meta-theorem of interest to us is soundness. This theorem says that every well-typed expression is *noninterferent*, i.e., the result of running an expression of a type labeled low is independent of substitutions used for its high-labeled free variables. This theorem is formalized below. Note that we work here with what is called termination-insensitive noninterference; we briefly discuss the termination-sensitive variant in Section 4.

Theorem 2.1 (Noninterference for FG)**.** *Suppose (1)* $\ell_i \not\sqsubseteq \ell$*, (2)* $x : \mathsf{A}^{\ell_i} \vdash_{pc} e : \mathsf{bool}^{\ell}$*, and (3)* $v_1, v_2 : \mathsf{A}^{\ell_i}$*. If both* $e[v_1/x]$ *and* $e[v_2/x]$ *terminate, then they produce the same value (of type* bool*).*

By definition, noninterference, as stated above is a relational (binary) property, i.e., it relates two runs of a program. Next, we show how to build a semantic model of FG's types that allows proving this property.

2.1.1. Semantic model of FG. We now describe our semantic model of FG's types. We use this model to show that the type system is sound (Theorem 2.1) and later to prove the soundness of our translations. Our semantic model uses the technique of step-indexed Kripke logical relations [18] and is more directly based on a model of types in a different domain, namely, incremental computational complexity [20]. In particular, our model captures all the invariants necessary to prove noninterference.

The central idea behind our model is to interpret each type in two different ways—as a set of values (unary interpretation), and as a set of pairs of values (binary interpretation). The binary interpretation is used to relate *low*-labeled types in the two runs mentioned in the noninterference

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}(e, x.e, x.e) \mid \mathsf{new}\ e \mid\ !e \mid e := e \\
\text{(Labeled) Types} & \tau & ::= & \mathsf{A}^{\ell} \\
\text{Unlabeled types} & \mathsf{A} & ::= & \mathsf{b} \mid \mathsf{unit} \mid \tau \xrightarrow{\ell_e} \tau \mid \tau \times \tau \mid \tau + \tau \mid \mathsf{ref}\ \tau \quad (\mathsf{b} \text{ denotes a base type})
\end{array}
$$

Typing judgment: $\boxed{\Gamma \vdash_{pc} e : \tau}$

$$\frac{}{\Gamma, x : \tau \vdash_{pc} x : \tau}\ \text{FG-var} \qquad \frac{\Gamma, x : \tau_1 \vdash_{\ell_e} e : \tau_2}{\Gamma \vdash_{pc} \lambda x.e : (\tau_1 \xrightarrow{\ell_e} \tau_2)^{\bot}}\ \text{FG-lam}$$

$$\frac{\Gamma \vdash_{pc} e_1 : (\tau_1 \xrightarrow{\ell_e} \tau_2)^{\ell} \qquad \Gamma \vdash_{pc} e_2 : \tau_1 \qquad \mathcal{L} \vdash \tau_2 \searrow \ell \qquad \mathcal{L} \vdash pc \sqcup \ell \sqsubseteq \ell_e}{\Gamma \vdash_{pc} e_1\ e_2 : \tau_2}\ \text{FG-app}$$

$$\frac{\Gamma \vdash_{pc} e_1 : \tau_1 \qquad \Gamma \vdash_{pc} e_2 : \tau_2}{\Gamma \vdash_{pc} (e_1, e_2) : (\tau_1 \times \tau_2)^{\bot}}\ \text{FG-prod} \qquad \frac{\Gamma \vdash_{pc} e : (\tau_1 \times \tau_2)^{\ell} \qquad \mathcal{L} \vdash \tau_1 \searrow \ell}{\Gamma \vdash_{pc} \mathsf{fst}(e) : \tau_1}\ \text{FG-fst}$$

$$\frac{\Gamma \vdash_{pc} e : \tau_1}{\Gamma \vdash_{pc} \mathsf{inl}(e) : (\tau_1 + \tau_2)^{\bot}}\ \text{FG-inl}$$

$$\frac{\Gamma \vdash_{pc} e : (\tau_1 + \tau_2)^{\ell} \qquad \Gamma, x : \tau_1 \vdash_{pc \sqcup \ell} e_1 : \tau \qquad \Gamma, y : \tau_2 \vdash_{pc \sqcup \ell} e_2 : \tau \qquad \mathcal{L} \vdash \tau \searrow \ell}{\Gamma \vdash_{pc} \mathsf{case}(e, x.e_1, y.e_2) : \tau}\ \text{FG-case}$$

$$\frac{\Gamma \vdash_{pc'} e : \tau' \qquad \mathcal{L} \vdash pc \sqsubseteq pc' \qquad \mathcal{L} \vdash \tau' <: \tau}{\Gamma \vdash_{pc} e : \tau}\ \text{FG-sub} \qquad \frac{\Gamma \vdash_{pc} e : \tau \qquad \mathcal{L} \vdash \tau \searrow pc}{\Gamma \vdash_{pc} \mathsf{new}\ e : (\mathsf{ref}\ \tau)^{\bot}}\ \text{FG-ref}$$

$$\frac{\Gamma \vdash_{pc} e : (\mathsf{ref}\ \tau)^{\ell} \qquad \mathcal{L} \vdash \tau <: \tau' \qquad \mathcal{L} \vdash \tau' \searrow \ell}{\Gamma \vdash_{pc} !e : \tau'}\ \text{FG-deref}$$

$$\frac{\Gamma \vdash_{pc} e_1 : (\mathsf{ref}\ \tau)^{\ell} \qquad \Gamma \vdash_{pc} e_2 : \tau \qquad \mathcal{L} \vdash \tau \searrow (pc \sqcup \ell)}{\Gamma \vdash_{pc} e_1 := e_2 : \mathsf{unit}}\ \text{FG-assign} \qquad \frac{}{\Gamma \vdash_{pc} () : \mathsf{unit}^{\bot}}\ \text{FG-unitI}$$

Subtyping judgments: $\boxed{\mathcal{L} \vdash \mathsf{A} <: \mathsf{A}'}$ and $\boxed{\mathcal{L} \vdash \tau <: \tau'}$

$$\frac{\mathcal{L} \vdash \ell \sqsubseteq \ell' \qquad \mathcal{L} \vdash \mathsf{A} <: \mathsf{A}'}{\mathcal{L} \vdash \mathsf{A}^{\ell} <: \mathsf{A}'^{\ell'}}\ \text{FGsub-label} \qquad \frac{}{\mathcal{L} \vdash \mathsf{ref}\ \tau <: \mathsf{ref}\ \tau}\ \text{FGsub-ref}$$

$$\frac{\mathcal{L} \vdash \tau_1' <: \tau_1 \qquad \mathcal{L} \vdash \tau_2 <: \tau_2' \qquad \mathcal{L} \vdash \ell_e' \sqsubseteq \ell_e}{\mathcal{L} \vdash \tau_1 \xrightarrow{\ell_e} \tau_2 <: \tau_1' \xrightarrow{\ell_e'} \tau_2'}\ \text{FGsub-arrow}$$

Figure 1. FG's language syntax and type system (selected rules)

theorem, while the unary interpretation is used to interpret *high*-labeled types independently in the two runs (since high-labeled values may be unrelated across the two runs). What is high and what is low is determined by the level of the observer (adversary), which is a parameter to our binary interpretation.

Remark. Readers familiar with earlier models of IFC type systems [7], [11], [16] may wonder why we need a unary relation, when prior work did not. The reason is that we handle an effect (mutable state) in our model, which prior work did not. In the absence of effects, the unary model is unnecessary. In the presence of effects, the unary relation captures what is often called the "confinement lemma" in proofs of noninterference—we need to know that while the two runs are executing high branches independently, neither will modify low-labeled locations.

Unary interpretation. The unary interpretation of types is shown in Figure 2. The interpretation is actually a Kripke model. It uses *worlds*, written θ, which specify the type for each valid (allocated) location in the heap. For example, $\theta(a) = \mathsf{bool}^{H}$ means that location a should hold a high boolean. The world can grow as the program executes and

$$
\begin{array}{lcl}
\lfloor \mathsf{b} \rfloor_V & \triangleq & \{(\theta, m, v) \mid v \in [\![\mathsf{b}]\!]\} \\
\lfloor \mathsf{unit} \rfloor_V & \triangleq & \{(\theta, m, v) \mid v \in [\![\mathsf{unit}]\!]\} \\
\lfloor \tau_1 \times \tau_2 \rfloor_V & \triangleq & \{(\theta, m, (v_1, v_2)) \mid (\theta, m, v_1) \in \lfloor \tau_1 \rfloor_V \wedge (\theta, m, v_2) \in \lfloor \tau_2 \rfloor_V\} \\
\lfloor \tau_1 + \tau_2 \rfloor_V & \triangleq & \{(\theta, m, \mathsf{inl}\ v) \mid (\theta, m, v) \in \lfloor \tau_1 \rfloor_V\} \cup \{(\theta, m, \mathsf{inr}\ v) \mid (\theta, m, v) \in \lfloor \tau_2 \rfloor_V\} \\
\lfloor \tau_1 \xrightarrow{\ell_e} \tau_2 \rfloor_V & \triangleq & \{(\theta, m, \lambda x.e) \mid \forall \theta'.\theta \sqsubseteq \theta' \wedge \forall j < m. \forall v.((\theta', j, v) \in \lfloor \tau_1 \rfloor_V \implies (\theta', j, e[v/x]) \in \lfloor \tau_2 \rfloor_E^{\ell_e})\} \\
\lfloor \mathsf{ref}\ \tau \rfloor_V & \triangleq & \{(\theta, m, a) \mid \theta(a) = \tau\} \\
\\
\lfloor \mathsf{A}^{\ell} \rfloor_V & \triangleq & \lfloor \mathsf{A} \rfloor_V
\end{array}
$$

$$
\begin{array}{lcl}
\lfloor \tau \rfloor_E^{pc} & \triangleq & \{(\theta, n, e) \mid \forall H.(n, H) \triangleright \theta \wedge \forall j < n.(H, e) \Downarrow_j (H', v') \implies \\
& & \quad \exists \theta'.\theta \sqsubseteq \theta' \wedge (n - j, H') \triangleright \theta' \wedge (\theta', n - j, v') \in \lfloor \tau \rfloor_V \wedge \\
& & \qquad (\forall a.H(a) \neq H'(a) \implies \exists \ell'.\theta(a) = \mathsf{A}^{\ell'} \wedge pc \sqsubseteq \ell') \wedge \\
& & \qquad (\forall a \in \mathtt{dom}(\theta') \backslash \mathtt{dom}(\theta).\theta'(a) \searrow pc)\}
\end{array}
$$

$$
(n, H) \triangleright \theta \quad \triangleq \quad \mathtt{dom}(\theta) \subseteq \mathtt{dom}(H) \wedge \forall a \in \mathtt{dom}(\theta).(\theta, n - 1, H(a)) \in \lfloor \theta(a) \rfloor_V
$$

Figure 2. Unary value, expression, and heap conformance relations for FG

$$
\begin{array}{lcl}
\lceil \mathsf{b} \rceil_V^{\mathcal{A}} & \triangleq & \{(W, n, v_1, v_2) \mid v_1 = v_2 \wedge \{v_1, v_2\} \in [\![\mathsf{b}]\!]\} \\
\lceil \mathsf{unit} \rceil_V^{\mathcal{A}} & \triangleq & \{(W, n, (), ()) \mid () \in [\![\mathsf{unit}]\!]\} \\
\lceil \tau_1 \times \tau_2 \rceil_V^{\mathcal{A}} & \triangleq & \{(W, n, (v_1, v_2), (v_1', v_2')) \mid (W, n, v_1, v_1') \in \lceil \tau_1 \rceil_V^{\mathcal{A}} \wedge (W, n, v_2, v_2') \in \lceil \tau_2 \rceil_V^{\mathcal{A}}\} \\
\lceil \tau_1 + \tau_2 \rceil_V^{\mathcal{A}} & \triangleq & \{(W, n, \mathsf{inl}\ v, \mathsf{inl}\ v') \mid (W, n, v, v') \in \lceil \tau_1 \rceil_V^{\mathcal{A}}\} \cup \\
& & \{(W, n, \mathsf{inr}\ v, \mathsf{inr}\ v') \mid (W, n, v, v') \in \lceil \tau_2 \rceil_V^{\mathcal{A}}\} \\
\lceil \tau_1 \xrightarrow{\ell_e} \tau_2 \rceil_V^{\mathcal{A}} & \triangleq & \{(W, n, \lambda x.e_1, \lambda x.e_2) \mid \\
& & \quad \forall W' \sqsupseteq W, j < n, v_1, v_2. \\
& & \qquad ((W', j, v_1, v_2) \in \lceil \tau_1 \rceil_V^{\mathcal{A}} \implies (W', j, e_1[v_1/x], e_2[v_2/x]) \in \lceil \tau_2 \rceil_E^{\mathcal{A}}) \wedge \\
& & \quad \forall \theta_l \sqsupseteq W.\theta_1, j, v_c.\ ((\theta_l, j, v_c) \in \lfloor \tau_1 \rfloor_V \implies (\theta_l, j, e_1[v_c/x]) \in \lfloor \tau_2 \rfloor_E^{\ell_e}) \wedge \\
& & \quad \forall \theta_l \sqsupseteq W.\theta_2, j, v_c.\ ((\theta_l, j, v_c) \in \lfloor \tau_1 \rfloor_V \implies (\theta_l, j, e_2[v_c/x]) \in \lfloor \tau_2 \rfloor_E^{\ell_e})\} \\
\lceil \mathsf{ref}\ \tau \rceil_V^{\mathcal{A}} & \triangleq & \{(W, n, a_1, a_2) \mid (a_1, a_2) \in W.\hat{\beta} \wedge W.\theta_1(a_1) = W.\theta_2(a_2) = \tau\} \\
\\
\lceil \mathsf{A}^{\ell} \rceil_V^{\mathcal{A}} & \triangleq & \begin{cases} \{(W, n, v_1, v_2) \mid (W, n, v_1, v_2) \in \lceil \mathsf{A} \rceil_V^{\mathcal{A}}\} & \ell \sqsubseteq \mathcal{A} \\ \{(W, n, v_1, v_2) \mid \forall i \in \{1, 2\}. \forall m.(W.\theta_i, m, v_i) \in \lfloor \mathsf{A} \rfloor_V\} & \ell \not\sqsubseteq \mathcal{A} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\lceil \tau \rceil_E^{\mathcal{A}} & \triangleq & \{(W, n, e_1, e_2) \mid \\
& & \quad \forall H_1, H_2, j < n.(n, H_1, H_2) \overset{\mathcal{A}}{\triangleright} W \wedge (H_1, e_1) \Downarrow_j (H_1', v_1') \wedge (H_2, e_2) \Downarrow (H_2', v_2') \implies \\
& & \qquad \exists W' \sqsupseteq W.(n - j, H_1', H_2') \overset{\mathcal{A}}{\triangleright} W' \wedge (W', n - j, v_1', v_2') \in \lceil \tau \rceil_V^{\mathcal{A}}\}
\end{array}
$$

$$
\begin{array}{lcl}
(n, H_1, H_2) \overset{\mathcal{A}}{\triangleright} W & \triangleq & \mathtt{dom}(W.\theta_1) \subseteq \mathtt{dom}(H_1) \wedge \mathtt{dom}(W.\theta_2) \subseteq \mathtt{dom}(H_2) \wedge (W.\hat{\beta}) \subseteq (\mathtt{dom}(W.\theta_1) \times \mathtt{dom}(W.\theta_2)) \wedge \\
& & \forall (a_1, a_2) \in (W.\hat{\beta}).(W.\theta_1(a_1) = W.\theta_2(a_2) \wedge (W, n - 1, H_1(a_1), H_2(a_2)) \in \lceil W.\theta_1(a_1) \rceil_V^{\mathcal{A}}) \wedge \\
& & \forall i \in \{1, 2\}. \forall m. \forall a_i \in \mathtt{dom}(W.\theta_i).(W.\theta_i, m, H_i(a_i)) \in \lfloor W.\theta_i(a_i) \rfloor_V
\end{array}
$$

Figure 3. Binary value, expression and heap conformance relations for FG

allocates more locations. A second important component used in the interpretation is a *step-index*, written m or n [21]. Step-indices are natural numbers, and are merely a technical device to break a non-well-foundedness issue in Kripke models of higher-order state, like this one. Our use of step-indices is standard and readers may ignore them.

The interpretation itself consists of three mutually inductive relations—a *value relation* for types (labeled and unlabeled), written $\lfloor\tau\rfloor_V$; an *expression relation* for labeled types, written $\lfloor\tau\rfloor_E^{pc}$; and a *heap conformance relation*, written $(n, H) \triangleright \theta$. These relations are well-founded by induction on the step indices n and types. This is the only role of step-indices in our model.

The value relation $\lfloor\tau\rfloor_V$ defines, for each type, which values (at which worlds and step-indices) lie in that type. For base types b, this is straightforward: All syntactic values of type b (written $[\![\mathsf{b}]\!]$) lie in $\lfloor\mathsf{b}\rfloor_V$ at any world and any step index. For pairs, the relation is the intuitive one: (v_1, v_2) is in $\lfloor\tau_1 \times \tau_2\rfloor_V$ iff v_1 is in $\lfloor\tau_1\rfloor_V$ and v_2 is in $\lfloor\tau_2\rfloor_V$. The function type $\tau_1 \xrightarrow{\ell_e} \tau_2$ contains a value $\lambda x.e$ at world θ if in any world θ' that extends θ, if v is in $\lfloor\tau_1\rfloor_V$, then $(\lambda x.e)\ v$ or, equivalently, $e[v/x]$, is in the *expression relation* $\lfloor\tau_2\rfloor_E^{\ell_e}$. We describe this expression relation below. Importantly, we allow for the world θ to be extended to θ' since between the time that the function $\lambda x.e$ was created and the time that the function is applied, new locations could be allocated. The type $\mathsf{ref}\ \tau$ contains all locations a whose type according to the world θ matches τ. Finally, security labels play no role in the unary interpretation, so $\lfloor\mathsf{A}^\ell\rfloor_V = \lfloor\mathsf{A}\rfloor_V$ (in contrast, labels play a significant role in the binary interpretation).

The expression relation $\lfloor\tau\rfloor_E^{pc}$ defines, for each type, which expressions lie in the type (at each pc, each world θ and each step index n). The definition may look complex, but is relatively straightforward: e is in $\lfloor\tau\rfloor_E^{pc}$ if for any heap H that conforms to the world θ such that running e starting from H results in a value v' and a heap H', there is a some extension θ' of θ to which H' conforms and at which v' is in $\lfloor\tau\rfloor_V$. Additionally, all writes performed during the execution (defined as the locations at which H and H' differ) must have labels above the program counter, pc. In simpler words, the definition simply says that e lies in $\lfloor\tau\rfloor_E^{pc}$ if its resulting value is in $\lfloor\tau\rfloor_V$, it preserves heap conformance with worlds and, importantly, its write effects are at labels above pc. (Readers familiar with proofs of noninterference should note that the condition on write effects is our model's analogue of the so-called "confinement lemma".)

The heap conformance relation $(n, H) \triangleright \theta$ defines when a heap H conforms to a world θ. The relation is simple; it holds when the heap H maps every location to a value in the semantic interpretation of the location's type given by the world θ.

Binary interpretation. The binary interpretation of types is shown in Figure 3. This interpretation relates two executions of a program with different inputs. Like the unary interpretation, this interpretation is also a Kripke model. Its worlds, written W, are different, though. Each world is a triple $W = (\theta_1, \theta_2, \hat{\beta})$. θ_1 and θ_2 are unary worlds that specify the types of locations allocated in the two executions. Since executions may proceed in sync on the two sides for a while, then diverge in a high-labeled branch, then possibly re-synchronize, and so on, some locations allocated on one side may have analogues on the other side, while other locations may be unique to either side. This is captured by $\hat{\beta}$, which is a *partial bijection* between the domains of θ_1 and θ_2. If $(a_1, a_2) \in \hat{\beta}$, then location a_1 in the first run corresponds to location a_2 in the second run. Any location not in $\hat{\beta}$ has no analogue on the other side.

As before, the interpretation itself consists of three mutually inductive relations—a *value relation* for types (labeled and unlabeled), written $\lceil\tau\rceil_V^{\mathcal{A}}$; an *expression relation* for labeled types, written $\lceil\tau\rceil_E^{\mathcal{A}}$; and a *heap conformance relation*, written $(n, H_1, H_2) \overset{\mathcal{A}}{\triangleright} W$. These relations are all parametrized by the level of the observer (adversary), $\mathcal{A}$, which is an element of $\mathcal{L}$.

The value relation $\lceil\tau\rceil_V^{\mathcal{A}}$ defines, for each type, which pairs of values from the two runs are related by that type (at each world, each step-index and each adversary). At base types, b, only identical values are related. For pairs, the relation is the intuitive one: (v_1, v_2) and (v_1', v_2') are related in $\lceil\tau_1 \times \tau_2\rceil_V^{\mathcal{A}}$ iff v_i and v_i' are related in $\lceil\tau_i\rceil_V^{\mathcal{A}}$ for $i \in \{1, 2\}$. Two values are related at a sum type only if they are both left injections or both right injections. At the function type $\tau_1 \xrightarrow{\ell_e} \tau_2$, two functions are related if they map values related at the argument type τ_1 to expressions related at the result type τ_2. For technical reasons, we also need both the functions to satisfy the conditions of the *unary* relation. At a reference type $\mathsf{ref}\ \tau$, two locations a_1 and a_2 are related at world $W = (\theta_1, \theta_2, \hat{\beta})$ only if they are related by $\hat{\beta}$ (i.e., they are correspondingly allocated locations) and their types as specified by θ_1 and θ_2 are equal to τ.

Finally, and most importantly, at a labeled type A^ℓ, $\lceil\mathsf{A}^\ell\rceil_V^{\mathcal{A}}$ relates values depending on the ordering between ℓ and the adversary $\mathcal{A}$. When $\ell \sqsubseteq \mathcal{A}$, the adversary can see values labeled ℓ, so $\lceil\mathsf{A}^\ell\rceil_V^{\mathcal{A}}$ contains exactly the values related in $\lceil\mathsf{A}\rceil_V^{\mathcal{A}}$. When $\ell \not\sqsubseteq \mathcal{A}$, values labeled ℓ are opaque to the adversary (in colloquial terms, they are "high"), so they can be arbitrary. In this case, $\lceil\mathsf{A}^\ell\rceil_V^{\mathcal{A}}$ is the cross product of the *unary* interpretation of A with itself. This is the only place in our model where the binary and unary interpretations interact.

The expression relation $\lceil\tau\rceil_E^{\mathcal{A}}$ defines, for each type, which pairs of expressions from the two executions lie in the type (at each world W, each step index n and each adversary $\mathcal{A}$). The definition is similar to that in the unary case: e_1 and e_2 lie in $\lceil\tau\rceil_E^{\mathcal{A}}$ if the values they produce are related in the value relation $\lceil\tau\rceil_V^{\mathcal{A}}$, and the expressions preserve heap conformance.

The heap conformance relation $(n, H_1, H_2) \overset{\mathcal{A}}{\triangleright} W$ defines when a pair of heaps H_1, H_2 conforms to a world $W = (\theta_1, \theta_2, \hat{\beta})$. The relation requires that any pair of locations related by $\hat{\beta}$ have the same types (according to θ_1 and θ_2), and that the values stored in H_1 and H_2 at these locations lie in the binary value relation of that common type.

Meta-theory. The primary meta-theoretic property of a logical relations model like ours is the so-called *fundamental theorem*. This theorem says that any expression syntactically in a type (as established via the type system) also lies in the semantic interpretation (the expression relation) of that type. Here, we have two such theorems—one for the unary interpretation and one for the binary interpretation.

To write these theorems, we define unary and binary interpretations of contexts, $\lfloor\Gamma\rfloor_V$ and $\lceil\Gamma\rceil^{\mathcal{A}}_V$, respectively. These interpretations specify when unary and binary substitutions conform to Γ. A unary substitution δ maps each variable to a value whereas a binary substitution γ maps each variable to two values, one for each run.

$$\lfloor\Gamma\rfloor_V \triangleq \{(\theta, n, \delta) \mid \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\delta) \wedge \forall x \in \mathtt{dom}(\Gamma).\ (\theta, n, \delta(x)) \in \lfloor\Gamma(x)\rfloor_V\}$$

$$\lceil\Gamma\rceil^{\mathcal{A}}_V \triangleq \{(W, n, \gamma) \mid \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\gamma) \wedge \forall x \in \mathtt{dom}(\Gamma).\ (W, n, \pi_1(\gamma(x)), \pi_2(\gamma(x))) \in \lceil\Gamma(x)\rceil^{\mathcal{A}}_V\}$$

Theorem 2.2 (Unary fundamental theorem)**.** *If* $\Gamma \vdash_{pc} e : \tau$ *and* $(\theta, n, \delta) \in \lfloor\Gamma\rfloor_V$*, then* $(\theta, n, e\ \delta) \in \lfloor\tau\rfloor^{pc}_E$.

Theorem 2.3 (Binary fundamental theorem)**.** *If* $\Gamma \vdash_{pc} e : \tau$ *and* $(W, n, \gamma) \in \lceil\Gamma\rceil^{\mathcal{A}}_V$*, then* $(W, n, e\ (\gamma{\downarrow_1}), e\ (\gamma{\downarrow_2})) \in \lceil\tau\rceil^{\mathcal{A}}_E$*, where* $\gamma{\downarrow_1}$ *and* $\gamma{\downarrow_2}$ *are the left and right projections of* γ.

The proofs of these theorems proceed by induction on the given derivations of $\Gamma \vdash_{pc} e : \tau$. The proofs are tedious, but not difficult or surprising. The primary difficulty, as with all logical relations models, is in setting up the model correctly, not in proving the fundamental theorems.

FG's noninterference theorem (Theorem 2.1) is a simple corollary of these two theorems.

2.2. The coarse-grained type system, CG

Our coarse-grained type system, CG, is a variant of the static fragment of the hybrid IFC type system HLIO [10].[4] Like FG, CG also operates on a higher-order, eager, call-by-value language with state, but it separates pure expressions from impure (stateful) ones at the level of types. This is done, as usual, by introducing a monad for state, and limiting all state-accessing operations (dereferencing, allocation, assignment) to the monad. However, in CG, as in HLIO, the monad also doubles as the unit of labeling. Values and types are not necessarily labeled individually in CG. Instead, there is a confidentiality label on an entire monadic computation. This makes CG coarse-grained.

CG's syntax and type system are shown in Figure 4. The types include all the usual types of the simply typed λ-calculus and, unlike FG, a label is not forced on every type. There are two special types: $\mathbb{C}\ \ell_1\ \ell_2\ \tau$ and $\mathsf{Labeled}\ \ell\ \tau$.

The type $\mathbb{C}\ \ell_1\ \ell_2\ \tau$ is the aforementioned monadic type of computations that may access the heap (expressions of other types cannot access the heap), eventually producing a value of type τ. The first label ℓ_1, called the *pc-label*, is a lower bound on the write effects of the computation. The second label ℓ_2, called the *taint label*, is an upper bound on the labels of all values that the computation has analyzed so far; it is, for this reason, also an *implicit* label on the output type τ of the computation, and on any intermediate values within the computation.

The type $\mathsf{Labeled}\ \ell\ \tau$ explicitly labels a value (of type τ) with label ℓ. In FG's notation, this would be analogous to τ^ℓ. The difference is that this labeling can be used *selectively* in CG; unlike FG, not every type must be labeled. Also, the reference type $\mathsf{ref}\ \ell\ \tau$ carries an explicit label ℓ in CG. Such a reference stores a value of type $\mathsf{Labeled}\ \ell\ \tau$. Labels on references are necessary to prevent implicit leaks via control dependencies—the type system relates the pc-label to the label of the written value at every assignment. Similar labels on references were unnecessary in FG since every written value always carries a label anyhow.

Typing rules. CG uses the standard typing judgment $\Gamma \vdash e : \tau$. There is no need for a pc on the judgment since effects are confined to the monad. CG uses the typing rules of the simply typed λ-calculus for the type constructs b, unit, $\times$, $+$ and $\rightarrow$. We do not re-iterate these standard rules, and focus here only on the new constructs. The construct $\mathsf{ret}(e)$ is the monadic return that immediately returns e, without any heap access. Consequently, it can be given the type $\mathbb{C}\ \top\ \bot\ \tau$ (rule CG-ret). The pc-label is $\top$ since the computation has no write effect, while the taint label is $\bot$ since the computation has not analyzed any value.

The monadic construct $\mathsf{bind}(e_1, x.e_2)$ sequences the computation e_2 after e_1, binding the return value of e_1 to x in e_2. The typing rule for this construct, CG-bind, is important and interesting. The rule says that $\mathsf{bind}(e_1, x.e_2)$ can be given the type $\mathbb{C}\ \ell\ \ell_4\ \tau'$ if $(e_1 : \mathbb{C}\ \ell_1\ \ell_2\ \tau)$, $(e_2 : \mathbb{C}\ \ell_3\ \ell_4\ \tau')$ and four conditions hold. The conditions $\ell \sqsubseteq \ell_1$ and $\ell \sqsubseteq \ell_3$ check that the pc-label of $\mathsf{bind}(e_1, x.e_2)$, which is ℓ, is below the pc-label of e_1 and e_2, which are ℓ_1 and ℓ_3, respectively. This ensures that the write effects of $\mathsf{bind}(e_1, x.e_2)$ are indeed above its pc-label, ℓ. The conditions $\ell_2 \sqsubseteq \ell_3$ and $\ell_2 \sqsubseteq \ell_4$ prevent leaking the output of e_1 via the write effects and the output of e_2, respectively. Observe how these conditions together track labels at the level of entire computations, i.e., coarsely.

Next, we describe rules pertaining to the type $\mathsf{Labeled}\ \ell\ \tau$. This type is introduced using the expression constructor Lb, as in rule CG-label. Dually, if $e : \mathsf{Labeled}\ \ell\ \tau$, then the construct $\mathsf{unlabel}(e)$ eliminates this label. This construct has the monadic type $\mathbb{C}\ \top\ \ell\ \tau$. The taint label ℓ indicates that the computation has (internally) analyzed something labeled ℓ. The pc-label is $\top$ since nothing has been written.

Rule CG-deref says that dereferencing (reading) a location of type $\mathsf{ref}\ \ell'\ \tau$ produces a computation of type $\mathbb{C}\ \top\ \bot\ (\mathsf{Labeled}\ \ell'\ \tau)$. The type is monadic because dereferencing accesses the heap. The value the computation returns is explicitly labeled at ℓ'. The pc-label is $\top$ since the computation does not write, while the taint label is $\bot$ since the computation does not analyze the value it reads from

4. Differences between CG and HLIO and their consequences are discussed in Section 4.

$$\begin{array}{llll} \text{Expressions} & e & ::= & x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}(e, x.e, y.e) \mid \mathsf{new}\ e \mid\ !e \mid e := e \mid () \mid \\ & & & \mathsf{Lb}(e) \mid \mathsf{unlabel}(e) \mid \mathsf{toLabeled}(e) \mid \mathsf{ret}(e) \mid \mathsf{bind}(e, x.e) \\ \text{Types} & \tau & ::= & \mathsf{b} \mid \mathsf{unit} \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \mathsf{ref}\ \ell\ \tau \mid \mathsf{Labeled}\ \ell\ \tau \mid \mathbb{C}\ \ell_1\ \ell_2\ \tau \end{array}$$

Typing judgment: $\boxed{\Gamma \vdash e : \tau}$

(All rules of the simply typed lambda-calculus pertaining to the types $\mathsf{b}, \tau \to \tau, \tau \times \tau, \tau + \tau$, and unit are included.)

$$\frac{\Gamma \vdash e_1 : \mathbb{C}\ \ell_1\ \ell_2\ \tau \qquad \Gamma, x : \tau \vdash e_2 : \mathbb{C}\ \ell_3\ \ell_4\ \tau' \qquad \ell \sqsubseteq \ell_1 \qquad \ell \sqsubseteq \ell_3 \qquad \ell_2 \sqsubseteq \ell_3 \qquad \ell_2 \sqsubseteq \ell_4}{\Gamma \vdash \mathsf{bind}(e_1, x.e_2) : \mathbb{C}\ \ell\ \ell_4\ \tau'}\ \text{CG-bind}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ret}(e) : \mathbb{C}\ \top\ \bot\ \tau}\ \text{CG-ret} \qquad \frac{\Gamma \vdash e : \tau' \qquad \mathcal{L} \vdash \tau' <: \tau}{\Gamma \vdash e : \tau}\ \text{CG-sub} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Lb}(e) : \mathsf{Labeled}\ \ell\ \tau}\ \text{CG-label}$$

$$\frac{\Gamma \vdash e : \mathsf{Labeled}\ \ell\ \tau}{\Gamma \vdash \mathsf{unlabel}(e) : \mathbb{C}\ \top\ \ell\ \tau}\ \text{CG-unlabel} \qquad \frac{\Gamma \vdash e : \mathsf{Labeled}\ \ell\ \tau}{\Gamma \vdash \mathsf{new}\ e : \mathbb{C}\ \ell\ \bot\ (\mathsf{ref}\ \ell\ \tau)}\ \text{CG-ref} \qquad \frac{\Gamma \vdash e : \mathsf{ref}\ \ell'\ \tau}{\Gamma \vdash !e : \mathbb{C}\ \top\ \bot\ (\mathsf{Labeled}\ \ell\ '\tau)}\ \text{CG-deref}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{ref}\ \ell\ \tau \qquad \Gamma \vdash e_2 : \mathsf{Labeled}\ \ell\ \tau}{\Gamma \vdash e_1 := e_2 : \mathbb{C}\ \ell\ \bot\ \mathsf{unit}}\ \text{CG-assign} \qquad \frac{\Gamma \vdash e : \mathbb{C}\ \ell\ \ell'\ \tau}{\Gamma \vdash \mathsf{toLabeled}(e) : \mathbb{C}\ \ell\ \bot\ (\mathsf{Labeled}\ \ell'\ \tau)}\ \text{CG-toLabeled}$$

Subtyping judgment: $\boxed{\mathcal{L} \vdash \tau <: \tau'}$

$$\frac{\mathcal{L} \vdash \tau <: \tau' \qquad \mathcal{L} \vdash \ell \sqsubseteq \ell'}{\mathcal{L} \vdash \mathsf{Labeled}\ \ell\ \tau <: \mathsf{Labeled}\ \ell'\ \tau'}\ \text{CGsub-labeled} \qquad \frac{\mathcal{L} \vdash \tau <: \tau' \qquad \mathcal{L} \vdash \ell_1' \sqsubseteq \ell_1 \qquad \mathcal{L} \vdash \ell_2 \sqsubseteq \ell_2'}{\mathcal{L} \vdash \mathbb{C}\ \ell_1\ \ell_2\ \tau <: \mathbb{C}\ \ell_1'\ \ell_2'\ \tau'}\ \text{CGsub-monad}$$

Figure 4. CG's language syntax and type system (selected rules)

the reference. (The taint label will change to ℓ' if the read value is subsequently unlabeled.) Dually, the rule CG-assign allows assigning a value labeled ℓ to a reference labeled ℓ. The result is a computation of type $\mathbb{C}\ \ell\ \bot\ \mathsf{unit}$. The pc-label ℓ indicates a write effect at level ℓ.

The last typing rule we highlight pertains to a special construct, $\mathsf{toLabeled}(e)$. This construct transforms e of monadic type $\mathbb{C}\ \ell\ \ell'\ \tau$ to the type $\mathbb{C}\ \ell\ \bot\ (\mathsf{Labeled}\ \ell'\ \tau)$. This is perfectly safe since the only way to observe the output of a monad is by binding its result and that result is explicitly labeled in the final type. The purpose of using this construct is to reduce the taint label of a computation to $\bot$. This allows a subsequent computation, which will *not* analyze the output of the current computation, to avoid raising its own taint label to ℓ'. Hence, this construct limits the scope of the taint label to a single computation, and prevents overtainting subsequent computations. We make extensive use of this construct in our translation from FG to CG. We note that HLIO's original typing rule for toLabeled is different, and does not always allow reducing the taint to $\bot$. We discuss the consequences of this difference in Section 4.

We briefly comment on subtyping for specific constructs in CG. Subtyping of $\mathsf{Labeled}\ \ell\ \tau$ is co-variant in ℓ, since it is always safe to increase a confidentiality label. Subtyping of $\mathbb{C}\ \ell_1\ \ell_2\ \tau$ is contra-variant in the pc-label ℓ_1 and co-variant in the taint label ℓ_2 since the former is a lower bound while the latter is an upper bound.

We prove soundness for CG by showing that every well-typed expression satisfies noninterference. Due to the presence of monadic types, the soundness theorem takes a specific form (shown below), and refers to a *forcing semantics*. These semantics operate on monadic types and actually perform reads and writes on the heap (in contrast, the pure evaluation semantics simply return suspended computations for monadic types). The forcing semantics are the expected ones, so we defer their details to the appendix.

Theorem 2.4 (Noninterference for CG)**.** *Suppose (1)* $\ell_i \not\sqsubseteq \ell$*, (2)* $x : \mathsf{Labeled}\ \ell_i\ \tau \vdash e : \mathbb{C}\ \_\ \ell\ \mathsf{bool}$*, and (3)* $v_1, v_2 : \mathsf{Labeled}\ \ell_i\ \tau$*. If both* $e[v_1/x]$ *and* $e[v_2/x]$ *terminate when forced, then they produce the same value (of type* bool*).*

2.2.1. Semantic model of CG. We build a semantic model of CG's types. The model is very similar in structure to the model of FG's types. We use two interpretations, unary and binary, and worlds exactly as in FG's model. The difference is that since state effects are confined to a monad in CG, all the constraints on heap updates move from the expression relations to the value relations at the monadic types. Owing to lack of space, and the similarity in the structures of the models, we defer CG's model and the fundamental theorems to the appendix.

3. Translations

In this section, we describe our translations from FG to CG and vice-versa, thus showing that these two type systems

are equally expressive. We start with the translation from FG to CG.

3.1. Translating FG to CG

Our goal in translating FG to CG is to show how a fine-grained IFC type system can be simulated in a coarse-grained one. Our translation is directed by the type derivations in FG and preserves typing and semantics. We describe the translation below, followed by formal properties of the translation. As a convention, we use the subscript or superscript s to indicate source (FG) elements, and t to indicate target (CG) elements. Thus e_s denotes a source expression, whereas e_t denotes a target expression.

The key idea of our translation is to map a source expression e_s satisfying $\vdash_{pc} e_s : \tau$ to a monadic target expression e_t satisfying $\vdash e_t : \mathbb{C}\ pc\ \bot\ (\!|\tau|\!)$. The pc used to type the source expression is mapped as-is to the pc-label of the monadic computation. The type of the source expression, τ, is translated by the function $(\!|\cdot|\!)$ that is described below. However—and this is the crucial bit—the taint label on the translated monadic computation is $\bot$. To get this $\bot$ taint we use the toLabeled construct judiciously. Not setting the taint to $\bot$ can cause a taint explosion in translated expressions, which would make it impossible to simulate the fine-grained dependence tracking of FG.

The function $(\!|\cdot|\!)$ defines how the types of source values are translated. This function is defined by induction on labeled and unlabeled source types.

$$
\begin{array}{lcl}
(\!|\mathsf{b}|\!) & = & \mathsf{b} \\
(\!|\mathsf{unit}|\!) & = & \mathsf{unit} \\
(\!|\tau_1 \xrightarrow{\ell_e} \tau_2|\!) & = & (\!|\tau_1|\!) \to \mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!) \\
(\!|\tau_1 \times \tau_2|\!) & = & (\!|\tau_1|\!) \times (\!|\tau_2|\!) \\
(\!|\tau_1 + \tau_2|\!) & = & (\!|\tau_1|\!) + (\!|\tau_2|\!) \\
(\!|\mathsf{ref}\ \tau|\!) & = & \mathsf{ref}\ \ell\ (\!|\mathsf{A}|\!) \qquad \text{when } \tau = \mathsf{A}^{\ell} \\
(\!|\mathsf{A}^{\ell}|\!) & = & \mathsf{Labeled}\ \ell\ (\!|\mathsf{A}|\!)
\end{array}
$$

The translation should be self-explanatory. The only non-trivial case is the translation of the function type $\tau_1 \xrightarrow{\ell_e} \tau_2$. A source function of this type is mapped to a target function that takes an argument of type $(\!|\tau_1|\!)$ and returns a monadic computation (the translation of the body of the source function) that has pc-label ℓ_e and eventually returns a value of type $(\!|\tau_2|\!)$.

Given this translation of types, we next define a type derivation-directed translation of expressions. This translation is formalized by the judgment $\Gamma \vdash_{pc} e_s : \tau \rightsquigarrow e_t$. The judgment means that translating the source expression e_s, which has the typing derivation $\Gamma \vdash_{pc} e_s$, yields the target expression e_t. This judgment is *functional*: For each type derivation $\Gamma \vdash_{pc} e_s : \tau$, it yields exactly one e_t. It is also easily implemented by induction on typing derivations. The rules for the judgment are shown in Figure 5. The thing to keep in mind while reading the rules is that e_t should have the type $\mathbb{C}\ pc\ \bot\ (\!|\tau|\!)$.

We illustrate how the translation works using one rule, FC-app. In this rule, we know inductively that the translation of e_1, i.e., e_{c1} has type $\mathbb{C}\ pc\ \bot\ (\!|(\tau_1 \xrightarrow{\ell_e} \tau_2)^{\ell}|\!)$, which is equal to $\mathbb{C}\ pc\ \bot\ (\mathsf{Labeled}\ \ell\ ((\!|\tau_1|\!) \to \mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!)))$. The translation of e_2, i.e., e_{c2} has type $\mathbb{C}\ pc\ \bot\ (\!|\tau_1|\!)$. We wish to construct something of type $\mathbb{C}\ pc\ \bot\ (\!|\tau_2|\!)$.

To do this, we bind e_{c1} to the variable a, which has the type $\mathsf{Labeled}\ \ell\ ((\!|\tau_1|\!) \to \mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!))$. Similarly, we bind e_{c2} to the variable b, which has the type $(\!|\tau_1|\!)$. Next, we unlabel a and bind the result to variable c, which has the type $(\!|\tau_1|\!) \to \mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!)$. However, due to the unlabeling, the *taint label on whatever computation we sequence after this bind must be at least* ℓ. Next, we apply b to c, which yields a value of type $\mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!)$. Via subtyping, using the assumption $pc \sqsubseteq \ell_e$, we can weaken this to $\mathbb{C}\ pc\ \ell\ (\!|\tau_2|\!)$. This satisfies the constraint that the taint label be at least ℓ and is *almost* what we need, except that we need the taint $\bot$ in place of ℓ.

To reduce the taint back to $\bot$, we use the *defined* CG function `coerce_taint`, which has the type $\mathbb{C}\ pc\ \ell\ \tau \to \mathbb{C}\ pc\ \bot\ \tau$, when τ has the form $\mathsf{Labeled}\ \ell'\ \tau'$ with $\ell \sqsubseteq \ell'$. This last constraint is satisfied here since we know that $\tau_2 \searrow \ell$. The function `coerce_taint` uses toLabeled internally and is defined in the figure.

This pattern of using `coerce_taint`, which internally contains toLabeled, to restrict the taint to $\bot$ is used to translate all elimination forms (application, projection, case, etc.). Overall, our translation uses toLabeled judiciously to prevent taint from exploding in the translated expressions.

Remark. Readers familiar with monads may note that our translation from FG to CG is based on the standard interpretation of the call-by-value λ-calculus in the computational λ-calculus [22]. Our translation additionally accounts for the pc and security labels, but is structurally the same.

Properties. Our translation preserves typing by construction. This is formalized in the following theorem. The context translation $(\!|\Gamma|\!)$ is defined pointwise on all types in Γ.

Theorem 3.1 (Typing preservation)**.** *If $\Gamma \vdash_{pc} e_s : \tau$ in FG, then there is a unique e_t such that $\Gamma \vdash_{pc} e_s : \tau \rightsquigarrow e_t$ and that e_t satisfies $(\!|\Gamma|\!) \vdash e_t : \mathbb{C}\ pc\ \bot\ (\!|\tau|\!)$ in CG.*

An immediate corollary of this theorem is that well-typed source programs translate to noninterfering target programs (since target typing implies noninterference in the target).

Next, we show that our translation preserves the meaning of programs, i.e., it is semantically "correct". For this, we define a *cross-language* logical relation, which relates source values (expressions) to target values (expressions) at each source type. This relation has three key properties: (A) A source expression and its translation are always in the relation (Theorem 3.2), (B) Related expressions reduce to related values, and (C) At base types, the relation is the identity. Together, these imply that our translation preserves the meanings of programs in the sense that a function from base types to base types maps to a target function with the same extension.

An excerpt of the relation is shown in Figure 6. The relation is defined over source (FG) types, and is divided

$$\frac{}{\Gamma, x : \tau \vdash_{pc} x : \tau \rightsquigarrow \mathsf{ret}\ x}\ \text{FC-var} \qquad \frac{\Gamma, x : \tau_1 \vdash_{\ell_e} e : \tau_2 \rightsquigarrow e_{c1}}{\Gamma \vdash_{pc} \lambda x.e : (\tau_1 \xrightarrow{\ell_\varsigma} \tau_2)^{\perp} \rightsquigarrow \mathsf{ret}(\mathsf{Lb}(\lambda x.e_{c1}))}\ \text{FC-lam}$$

$$\frac{\Gamma \vdash_{pc} e_1 : (\tau_1 \xrightarrow{\ell_\varsigma} \tau_2)^{\ell} \rightsquigarrow e_{c1} \qquad \Gamma \vdash_{pc} e_2 : \tau_1 \rightsquigarrow e_{c2} \qquad \mathcal{L} \vdash \ell \sqcup pc \sqsubseteq \ell_e \qquad \mathcal{L} \vdash \tau_2 \searrow \ell}{\Gamma \vdash_{pc} e_1\ e_2 : \tau_2 \rightsquigarrow \texttt{coerce\_taint}(\mathsf{bind}(e_{c1}, a.\mathsf{bind}(e_{c2}, b.\mathsf{bind}(\mathsf{unlabel}\ a, c.(c\ b)))))}\ \text{FC-app}$$

$$\frac{\Gamma \vdash_{pc} e_1 : \tau_1 \rightsquigarrow e_{c1} \qquad \Gamma \vdash_{pc} e_2 : \tau_2 \rightsquigarrow e_{c2}}{\Gamma \vdash_{pc} (e_1, e_2) : (\tau_1 \times \tau_2)^{\perp} \rightsquigarrow \mathsf{bind}(e_{c1}, a.\mathsf{bind}(e_{c2}, b.\mathsf{ret}(\mathsf{Lb}(a, b))))}\ \text{FC-prod}$$

$$\frac{\Gamma \vdash_{pc} e : (\tau_1 \times \tau_2)^{\ell} \rightsquigarrow e_c \qquad \mathcal{L} \vdash \tau_1 \searrow \ell}{\Gamma \vdash_{pc} \mathsf{fst}(e) : \tau_1 \rightsquigarrow \texttt{coerce\_taint}(\mathsf{bind}(e_c, a.\mathsf{bind}(\mathsf{unlabel}\ a, b.\mathsf{ret}(\mathsf{fst}(b)))))}\ \text{FC-fst}$$

$$\frac{\Gamma \vdash_{pc} e : \tau_1 \rightsquigarrow e_c}{\Gamma \vdash_{pc} \mathsf{inl}(e) : (\tau_1 + \tau_2)^{\perp} \rightsquigarrow \mathsf{bind}(e_c, a.\mathsf{ret}(\mathsf{Lb}(\mathsf{inl}(a))))}\ \text{FC-inl}$$

$$\frac{\Gamma \vdash_{pc} e : (\tau_1 + \tau_2)^{\ell} \rightsquigarrow e_c \qquad \Gamma, x : \tau_1 \vdash_{pc \sqcup \ell} e_1 : \tau \rightsquigarrow e_{c1} \qquad \Gamma, x : \tau_1 \vdash_{pc \sqcup \ell} e_2 : \tau \rightsquigarrow e_{c2} \qquad \mathcal{L} \vdash \tau \searrow \ell}{\Gamma \vdash_{pc} \mathsf{case}(e, x.e_1, y.e_2) : \tau \rightsquigarrow \texttt{coerce\_taint}(\mathsf{bind}(e_c, a.\mathsf{bind}(\mathsf{unlabel}\ a, b.\mathsf{case}(b, x.e_{c1}, y.e_{c2}))))}\ \text{FC-case}$$

$$\frac{\Gamma \vdash_{pc} e : (\mathsf{ref}\ \tau)^{\ell} \rightsquigarrow e_c \qquad \mathcal{L} \vdash \tau <: \tau' \qquad \mathcal{L} \vdash \tau' \searrow \ell}{\Gamma \vdash_{pc} !e : \tau \rightsquigarrow \texttt{coerce\_taint}(\mathsf{bind}(e_c, a.\mathsf{bind}(\mathsf{unlabel}\ a, b.!b)))}\ \text{FC-deref}$$

$$\frac{\Gamma \vdash_{pc} e_1 : (\mathsf{ref}\ \tau)^{\ell} \rightsquigarrow e_{c1} \qquad \Gamma \vdash_{pc} e_2 : \tau \rightsquigarrow e_{c2} \qquad \tau \searrow (pc \sqcup \ell)}{\Gamma \vdash_{pc} e_1 := e_2 : \mathsf{unit} \rightsquigarrow \mathsf{bind}(\mathsf{toLabeled}(\mathsf{bind}(e_{c1}, a.\mathsf{bind}(e_{c2}, b.\mathsf{bind}(\mathsf{unlabel}\ a, c.c := b)))), d.\mathsf{ret}())}\ \text{FC-assign}$$

where,
$$\texttt{coerce\_taint} : \mathbb{C}\ pc\ \ell\ \tau \to \mathbb{C}\ pc\ \perp\ \tau \qquad \text{when } \tau = \mathsf{Labeled}\ \ell'\ \tau' \text{ and } \ell \sqsubseteq \ell'$$
$$\texttt{coerce\_taint} \triangleq \lambda x.\mathsf{toLabeled}(\mathsf{bind}(x, y.\mathsf{unlabel}\ y))$$

Figure 5. Expression translation FG to CG (selected rules only)

$$\begin{array}{lcl}
\lfloor \mathsf{b} \rfloor_V^{\hat{\beta}} & \triangleq & \{({}^s\theta, m, {}^s v, {}^t v) \mid {}^s v \in [\![\mathsf{b}]\!] \wedge {}^t v \in [\![\mathsf{b}]\!] \wedge {}^s v = {}^t v\} \\
\lfloor \tau_1 \xrightarrow{\ell_\varsigma} \tau_2 \rfloor_V^{\hat{\beta}} & \triangleq & \{({}^s\theta, m, \lambda x.e_s, \lambda x.e_t) \mid \forall {}^s\theta' \sqsupseteq {}^s\theta, {}^s v, {}^t v, j < m, \hat{\beta} \sqsubseteq \hat{\beta}'.({}^s\theta', j, {}^s v, {}^t v) \in \lfloor \tau_1 \rfloor_V^{\hat{\beta}'} \implies \\
& & \qquad ({}^s\theta', j, e_s[{}^s v/x], e_t[{}^t v/x]) \in \lfloor \tau_2 \rfloor_E^{\hat{\beta}'}\} \\
\lfloor \mathsf{ref}\ \tau \rfloor_V^{\hat{\beta}} & \triangleq & \{({}^s\theta, m, a_s, a_t) \mid {}^s\theta(a_s) = \tau \wedge ({}^s a, {}^t a) \in \hat{\beta}\} \\
\lfloor \mathsf{A}^{\ell'} \rfloor_V^{\hat{\beta}} & \triangleq & \{({}^s\theta, m, {}^s v, \mathsf{Lb}({}^t v)) \mid ({}^s\theta, m, {}^s v, {}^t v) \in \lfloor \mathsf{A} \rfloor_V^{\hat{\beta}}\}
\end{array}$$

$$\begin{array}{lcl}
\lfloor \tau \rfloor_E^{\hat{\beta}} & \triangleq & \{({}^s\theta, n, e_s, e_t) \mid \forall H_s, H_t.(n, H_s, H_t) \overset{\hat{\beta}}{\triangleright} {}^s\theta \wedge \forall i < n, {}^s v.(H_s, e_s) \Downarrow_i (H_s', {}^s v) \implies \\
& & \qquad \exists H_t', {}^t v.(H_t, e_t) \Downarrow^f (H_t', {}^t v) \wedge \exists {}^s\theta' \sqsupseteq {}^s\theta, \hat{\beta}' \sqsupseteq \hat{\beta}.(n - i, H_s', H_t') \overset{\hat{\beta}'}{\triangleright} {}^s\theta' \\
& & \qquad \wedge ({}^s\theta', n - i, {}^s v, {}^t v) \in \lfloor \tau \rfloor_V^{\hat{\beta}'}\}
\end{array}$$

Figure 6. Cross-language value and expression relations for the FG to CG translation (excerpt)

(like our earlier relations) into a value relation $\lfloor\cdot\rfloor_V^{\hat{\beta}}$, an expression relation $\lfloor\cdot\rfloor_E^{\hat{\beta}}$, and a heap relation $(n, H_s, H_t) \overset{\hat{\beta}}{\triangleright} {}^s\theta$, which we omit here. The relations specify when a source value (resp. expression, heap) is related to a target value (resp. expression, heap) at a source unary world ${}^s\theta$, a step index n and a partial bijection $\hat{\beta}$ that relates source locations to corresponding target locations. The relation actually mirrors the unary logical relation for FG. The definition of the expression relation forces property (B) above, while the value relation at base types forces property (C).

Our main result is again a fundamental theorem, shown below. The symbols δ^s and δ^t denote unary substitutions in the source and target, respectively. The relation $\lfloor\Gamma\rfloor_V^{\hat{\beta}}$ is the obvious one, obtained by pointwise lifting of the value relation; its definition is in the appendix.

Theorem 3.2 (Fundamental theorem)**.** *If* $\Gamma \vdash_{pc} e_s : \tau \rightsquigarrow e_t$ *and* $({}^s\theta, n, \delta^s, \delta^t) \in \lfloor\Gamma\rfloor_V^{\hat{\beta}}$*, then* $({}^s\theta, n, e_s\ \delta^s, e_t\ \delta^t) \in \lfloor\tau\rfloor_E^{\hat{\beta}}$.

The proof of this theorem is by induction on the derivation of $\Gamma \vdash_{pc} e_s : \tau \rightsquigarrow e_t$. This theorem has two important consequences. First, it immediately implies property (A) above and, hence, completes the argument that our translation is semantically correct. Second, the theorem, along with the binary fundamental theorem for CG, allows us to re-derive the noninterference theorem for FG (Theorem 2.1) directly. This re-derivation is important because it provides confidence that our translation preserves the meaning of security labels. As a simple counterexample, it is perfectly possible to translate FG programs to CG programs, preserving both typing and semantics, by mapping all source labels to the same target label (say, $\bot$). However, we would not be able to re-derive the source noninterference theorem using the target's properties if this were the case.

3.2. Translating CG to FG

This section describes the translation in the other direction—from CG to FG. The overall structure (but not the details!) of this translation are similar to that of the earlier FG to CG translation, so we skip some boilerplate material here. The superscript or subscript s (source) now marks elements of CG and t (target) marks elements of FG.

The key idea of the translation is to map a source (CG) expression e_s satisfying $\vdash e_s : \tau$ to a target (FG) expression e_t satisfying $\vdash_\top e_t : [\![\tau]\!]$. The type translation $[\![\tau]\!]$ is defined below. The pc for the translated expression is $\top$ because, in CG, all effects are confined to a monad, so at the top-level, there are no effects. In particular, there are no write effects, so we can pick any pc; we pick the most informative pc, $\top$.

The type translation, $[\![\tau]\!]$, is defined by induction on τ.

$$
\begin{array}{ll}
[\![\mathsf{b}]\!] & = \mathsf{b}^\bot \\
[\![\tau_1 \to \tau_2]\!] & = ([\![\tau_1]\!] \overset{\top}{\to} [\![\tau_2]\!])^\bot \\
[\![\tau_1 \times \tau_2]\!] & = ([\![\tau_1]\!] \times [\![\tau_2]\!])^\bot \\
[\![\tau_1 + \tau_2]\!] & = ([\![\tau_1]\!] + [\![\tau_2]\!])^\bot \\
[\![\mathsf{ref}\ \ell\ \tau]\!] & = (\mathsf{ref}\ ([\![\tau]\!] + \mathsf{unit})^\ell)^\bot \\
[\![\mathbb{C}\ \ell_1\ \ell_2\ \tau]\!] & = (\mathsf{unit} \overset{\ell_1}{\to} ([\![\tau]\!] + \mathsf{unit})^{\ell_2})^\bot \\
[\![\mathsf{Labeled}\ \ell\ \tau]\!] & = ([\![\tau]\!] + \mathsf{unit})^\ell
\end{array}
$$

The most interesting case of the translation is that for $\mathbb{C}\ \ell_1\ \ell_2\ \tau$. Since a CG value of this type is a suspended computation, we map this type to a *thunk*—a suspended computation implemented as a function whose argument has type unit. The pc-label on the function matches the pc-label ℓ_1 of the source type. The taint label ℓ_2 is placed on the output type $[\![\tau]\!]$ using a coding trick: $([\![\tau]\!] + \mathsf{unit})^{\ell_2}$. The expression translation of monadic expressions only ever produces values labeled inl, so the right type of the sum, unit, is never reached during the execution of a translated expression. The same coding trick is used to translate labeled and ref types. We could also have used a different coding in place of $([\![\tau]\!] + \mathsf{unit})^{\ell_2}$. For example, $([\![\tau]\!] \times \mathsf{unit})^{\ell_2}$ works equally well.

The expression translation is directed by source typing derivations and is defined by the judgment $\Gamma \vdash e_s : \tau \rightsquigarrow e_t$, some of whose rules are shown in Figure 7. The translation is fairly straightforward (given the type translation). The only noteworthy aspect is the use of the injection inl wherever an expression of the type form $([\![\tau]\!] + \mathsf{unit})^\ell$ needs to be constructed.

Properties. The translation preserves typing by construction, as formalized in the following theorem. The context translation $[\![\Gamma]\!]$ is defined pointwise on all types in Γ.

Theorem 3.3 (Typing preservation)**.** *If* $\Gamma \vdash e_s : \tau$ *in CG, then there is a unique* e_t *such that* $\Gamma \vdash e_s : \tau \rightsquigarrow e_t$ *and that* e_t *satisfies* $[\![\Gamma]\!] \vdash_\top e_t : [\![\tau]\!]$ *in FG.*

Again, a corollary of this theorem is that well-typed source programs translate to noninterfering target programs.

We further prove that the translation preserves the semantics of programs. Our approach is the same as that for the FG to CG translation—we set up a cross-language logical relation, this time indexed by CG types, and show the fundamental theorem. From this, we derive that the translation preserves the meanings of programs. Additionally, we derive the noninterference theorem for CG using the binary fundamental theorem of FG, thus gaining confidence that our translation maps security labels properly. Since this development mirrors that for our earlier translation, we defer the details to the appendix.

4. Discussion

Practical implications. Our results establish that a coarse-grained IFC type system that labels at the granularity of entire computations can be as expressive as a fine-grained

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow e_F}{\Gamma \vdash \mathsf{Lb}_\ell(e) : \mathsf{Labeled}\ \ell\ \tau \rightsquigarrow \mathsf{inl}(e_F)}\ \text{label} \qquad \frac{\Gamma \vdash e : \mathsf{Labeled}\ \ell\ \tau \rightsquigarrow e_F}{\Gamma \vdash \mathsf{unlabel}(e) : \mathbb{C}\ \top\ \ell\ \tau \rightsquigarrow \lambda\_.e_F}\ \text{unlabel}$$

$$\frac{\Gamma \vdash e : \mathbb{C}\ \ell_1\ \ell_2\ \tau \rightsquigarrow e_F}{\Gamma \vdash \mathsf{toLabeled}(e) : \mathbb{C}\ \ell_1\ \bot\ (\mathsf{Labeled}\ \ell_2\ \tau) \rightsquigarrow \lambda\_.\mathsf{inl}(e_F\ ())}\ \text{toLabeled} \qquad \frac{\Gamma \vdash e : \tau \rightsquigarrow e_F}{\Gamma \vdash \mathsf{ret}(e) : \mathbb{C}\ \top\ \bot\ \tau \rightsquigarrow \lambda\_.\mathsf{inl}(e_F)}\ \text{ret}$$

$$\frac{\Gamma \vdash e_1 : \mathbb{C}\ \ell_1\ \ell_2\ \tau \rightsquigarrow e_{F1} \qquad \Gamma, x : \tau \vdash e_2 : \mathbb{C}\ \ell_3\ \ell_4\ \tau' \rightsquigarrow e_{F2} \qquad \ell \sqsubseteq \ell_1 \qquad \ell \sqsubseteq \ell_3 \qquad \ell_2 \sqsubseteq \ell_3 \qquad \ell_2 \sqsubseteq \ell_4}{\Gamma \vdash \mathsf{bind}(e_1, x.e_2) : \mathbb{C}\ \ell\ \ell_4\ \tau' \rightsquigarrow \lambda\_.\mathsf{case}(e_{F1}(), x.e_{F2}(), y.\mathsf{inr}())}\ \text{bind}$$

Figure 7. Expression translation CG to FG (selected rules only)

IFC type system that labels every individual value, if the coarse-grained type system has a construct like toLabeled to limit the scope of taints. It is also usually the case that a coarse-grained type system burdens a programmer less with annotations as compared to a fine-grained type system. This leads to the conclusion that, in general, there is merit to preferring coarse-grained IFC type systems with taint-scope limiting constructs over fine-grained IFC type systems. In a coarse-grained type system, the programmer can benefit from the reduced annotation burden and simulate the fine-grained type system when the fine-grained labeling is absolutely necessary for verification. Since our embedding of the fine-grained type system in the coarse-grained type system is compositional, it can be easily implemented in the coarse-grained type system as a library of macros, one for each construct of the language of the fine-grained type system.

Original HLIO. The original HLIO system [10], from whose static fragment CG is adapted, differs from CG in the interpretation of the labels ℓ_1 and ℓ_2 in the monadic type $\mathbb{C}\ \ell_1\ \ell_2\ \tau$. CG interprets these labels as the pc-label and the taint label, respectively. HLIO interprets these labels as the *starting taint* and the *ending taint* of the computation. This implies an *invariant* that $\ell_1 \sqsubseteq \ell_2$ and makes HLIO more restrictive that CG. The relevant consequence of this difference is that the rule for toLabeled cannot always lower the final taint to $\bot$. HLIO's rule is:

$$\frac{\Gamma \vdash e : \mathbb{C}\ \ell\ \ell'\ \tau}{\Gamma \vdash \mathsf{toLabeled}(e) : \mathbb{C}\ \ell\ \ell\ (\mathsf{Labeled}\ \ell'\ \tau)}\ \text{CG-toLabeled}$$

This restrictive rule makes it impossible to translate from FG to HLIO in the way we translate from FG to CG. In fact, [15] already explains how this restriction makes a translation from FG to the static fragment of HLIO very difficult. Our observation here is that HLIO's restriction, inherited from a prior system called LIO, is not important for statically enforced IFC and eliminating it allows a simple embedding of a fine-grained IFC type system.

Nonetheless, we did investigate further whether we can embed FG into the static fragment of the unmodified HLIO. The answer is still affirmative, but the embedding is complex and requires nontrivial quantification over labels. Part II of our appendix contains a complete account of this embedding (in fact, the appendix contains a parallel account of all our results using the original HLIO in place of CG).

HLIO also has two constructs, getLabel and labelOf, that allow reflection on labels. However, these constructs are meaningful only because HLIO uses hybrid (both static and dynamic) enforcement and carries labels at runtime. In a purely static enforcement, such as CG's, labels are not carried at runtime, so reflection on them is not meaningful.

Full abstraction. Since our translations preserve typedness, they map well-typed source programs to noninterfering target programs. However, an open question is whether they preserve contextual equivalence, i.e., whether they are fully abstract. Establishing full abstraction will allow translated source expressions to be freely co-linked with target expressions. We haven't attempted a proof of full abstraction yet, but it looks like an interesting next step. We note that since our dynamic semantics (big-step evaluation) are not cognizant of IFC (which is enforced completely statically), it may be sufficient to generalize our translations to simply-typed variants of FG and CG, and prove those fully abstract.

Other IFC properties. Our current setup is geared towards proving *termination-insensitive* noninterference, where the adversary cannot observe nontermination. We believe that the approach itself and the equivalence result should generalize to termination-sensitive noninterference, but will require nontrivial changes to our development. For example, we will have to change our binary logical relations to imply co-termination of related expressions and, additionally, modify the type systems to track nontermination as a separate effect.

Another relevant question in whether our equivalence result can be extended to type systems that support declassification and, more foundationally, whether our logical relations can handle declassification. This is a nuanced question, since it is unclear hitherto how declassification can be given a compositional semantic model. We are working on this problem currently.

5. Related work

We focus on related work directly connected to our contributions—logical relations for IFC type systems and language translations that care about IFC.

Logical relations for IFC type systems. Logical relations for IFC type systems have been studied before to

a limited extent. Sabelfeld and Sands develop a general theory of models of information flow types based on partial-equivalence relations (PERs), the mathematical foundation of logical relations [16]. However, they do not use these models for proving any specific type system or translation sound. The pure fragment of the SLam calculus was proven sound (in the sense of noninterference) using a logical relations argument [7, Appendix A]. However, to the best of our knowledge, the relation and the proof were not extended to mutable state. The proof of noninterference for Flow Caml [3], which is very close to SLam, considers higher-order state (and exceptions), but the proof is syntactic, not based on logical relations. The dependency core calculus (DCC) [11] also has a logical relations model but, again, the calculus is pure. The DCC paper also includes a state-passing embedding from the IFC type system of Volpano, Irvine and Smith [4], but the state is first-order. Mantel *et al.* use a security criterion based on an indistinguishability relation that is a PER to prove the soundness of a flow-sensitive type system for a concurrent language [17]. Their proof is also semantic, but the language is first-order. In contrast to these prior pieces of work, our logical relations handle higher-order state, and this complicates the models substantially; we believe we are the first to do so in the context of IFC.

Our models are based on the now-standard step-indexed Kripke logical relations [18], which have been used extensively for showing the soundness of program verification logics. Our model for FG is directly inspired by Cicek *et al.*'s model for a pure calculus of incremental programs [20]. That calculus does not include state, but the model is structurally very similar to our model of FG in that it also uses a unary and a binary relation that interact at labeled types. Extending that model with state was a significant amount of work, since we had to introduce Kripke worlds. Our model for CG has no direct predecessor; we developed it using ideas from our model of FG. (DCC is also coarse-grained and uses a labeled monad to track dependencies, but its model is quite different from ours in the treatment of the monadic type.)

Language translations that care about IFC. Language translations that preserve information flow properties appear in the DCC paper. The translations start from SLam's pure fragment and the type system of Volpano, Irvine and Smith and go into DCC. The paper also shows how to recover the noninterference theorem of the source of a translation from properties of the target, a theorem we also prove for our translations. Barthe *et al.* [19] describe a compilation from a high-level imperative language to a low-level assembly-like language. They show that their compilation is type and semantics preserving. They also derive noninterference for the source from the noninterference of the target. Fournet and Rezk [23] describe a compilation from an IFC-typed language to a low-level language where confidentiality and integrity are enforced using cryptography. They prove that well-typed source programs compile to noninterfering target programs, where the target noninterference is defined in a computational sense. Algehed and Russo [24] define an embedding of DCC into Haskell. They also consider an extension of DCC with state but, to the best of our knowledge, they do not prove any formal properties of the translation.

6. Conclusion

This paper has examined the question of whether information flow type systems that label at fine granularity and those that label at coarse granularity are equally expressive. We answer this question in the affirmative, assuming that the coarse-grained type system has a construct to limit the scope of the taint label. A more foundational contribution of our work is a better understanding of semantic models of information flow types. To this end, we have presented logical relations models of types in both the fine-grained and the coarse-grained settings, for calculi with mutable higher-order state.

Acknowledgments. This work was partly supported by the German Science Foundation (DFG) through the project "Information Flow Control for Browser Clients – IFC4BC" in the priority program "Reliably Secure Software Systems – RS^3", and also through the Collaborative Research Center "Methods and Tools for Understanding and Controlling Privacy" (SFB 1223). We thank our anonymous reviewers and anonymous shepherd for their helpful feedback.

References

[1] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, (PLAS)*, 2009.

[2] ——, "Permissive dynamic information flow analysis," in *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, (PLAS)*, 2010.

[3] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 1, 2003.

[4] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security (JCS)*, vol. 4, no. 2/3, 1996.

[5] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 4, 2000.

[6] N. Broberg, B. Delft, and D. Sands, "Paragon for practical programming with information-flow control," in *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, 2013.

[7] N. Heintze and J. G. Riecke, "The SLam calculus: Programming with secrecy and integrity," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1998.

[8] A. A. Matos and G. Boudol, "On declassification and the non-disclosure policy," *Journal of Computer Security (JCS)*, vol. 17, no. 5, 2009.

[9] G. Boudol, "Secure information flow as a safety property," in *International Workshop on Formal Aspects in Security and Trust (FAST)*, 2008.

[10] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: Mixing static and dynamic typing for information-flow control in Haskell," in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2015.

[11] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999.

[12] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the IEEE Symposium on Security and Privacy*, 1982.

[13] M. Felleisen, "On the expressive power of programming languages," *Science of Computer Programming*, vol. 17, no. 1-3, 1991.

[14] S. Hunt and D. Sands, "On flow-sensitive security types," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006.

[15] V. Rajani, I. Bastys, W. Rafnsson, and D. Garg, "Type systems for information flow control: The question of granularity," *SIGLOG News*, vol. 4, no. 1, 2017.

[16] A. Sabelfeld and D. Sands, "A PER model of secure information flow in sequential programs," in *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)*, 1999.

[17] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, 2011.

[18] A. Ahmed, D. Dreyer, and A. Rossberg, "State-dependent representation independence," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009.

[19] G. Barthe, T. Rezk, and A. Basu, "Security types preserving compilation," *Computer Languages, Systems & Structures (CLSS)*, vol. 33, no. 2, 2007.

[20] E. Çiçek, Z. Paraskevopoulou, and D. Garg, "A type theory for incremental computational complexity with control flow changes," in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2016.

[21] A. J. Ahmed, "Step-indexed syntactic logical relations for recursive and quantified types," in *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)*, 2006.

[22] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, 1991.

[23] C. Fournet and T. Rezk, "Cryptographically sound implementations for typed information-flow security," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008.

[24] M. Algehed and A. Russo, "Encoding DCC in Haskell," in *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, (PLAS)*, 2017.