Journal of Computer Security 0 (2020) 1–0
IOS Press

On the Expressiveness and Semantics of Information Flow Types

Vineet Rajani * and Deepak Garg
Max Planck Institute for Software Systems, Germany
E-mails: vrajani@mpi-sws.org, dg@mpi-sws.org

Abstract. Information Flow Control (IFC) is a form of dependence analysis that tracks and prohibits dependence of public outputs on secret inputs. Such a dependence analysis is often carried out using a type system. IFC type systems can track dependence (via confidentiality labels) at varying levels of granularity. On one extreme, there are fine-grained type systems that track dependence at the level of individual values. They label individual values. On the other extreme, there are coarse-grained type systems that track dependence at the level of entire computations. These type systems do not label individual values but instead label entire sub-computations.

An important foundational question is one of the relative expressiveness of these two classes of IFC type systems. In this paper we show that, despite the glaring differences in how they track dependence, the two classes of type systems are actually equally expressive. We do this by showing translations from FG, a fine-grained IFC type system derived from SLAM [1], to SLIO*, a coarse-grained IFC type system derived from HLIO [2], and vice-versa. The translation from SLIO* to FG is straightforward since FG tracks dependence at a granularity finer than SLIO* does. However, the translation from FG to SLIO* is quite involved and relies extensively on label quantification. We further examine the reason for this complexity using a slight variant of SLIO*, called CG, to which FG can be translated more easily.

As a separate, more foundational contribution we show how to extend logical relation models of information flow to languages with higher-order state. Specifically, we build world-indexed (Kripke) logical relations for FG, SLIO* and CG, which we use to prove these type systems sound and also to prove the translations between them correct.

Keywords: Information flow control, Type systems, Typed translation, Logical relations, Type system expressiveness

1. Introduction

Information Flow Control (IFC) is a technique for tracking flows of information between different elements of a computer system. This is often used to prevent illicit flows with respect to a relevant security policy. In a language-based setting, IFC can be performed dynamically using runtime monitoring [3, 4] or statically using type systems [1, 2, 5–11] for instance. In this paper, we focus only on the latter, i.e., on type-based approaches for IFC.

IFC type systems use security labels (elements of a security lattice) for tracking and prohibiting undesired flows of information. There are two significant aspects of security labels that govern the *expressiveness* of a sound type system, i.e., how many secure programs the type system can accept.[1] The first aspect is the granularity at which labels are specified. For instance, a type system with fine-grained labels might be able to specify the precise variables or inputs on which a value depends. A type system with

*Corresponding author. E-mail: vrajani@mpi-sws.org.

[1] Since checking correctness of IFC is undecidable by Rice's theorem, no IFC type system can be both sound and complete, i.e., accept exactly all secure programs.

coarser labels might abstract that information to a coarser classification like "secret" and "public". The effect of varying the granularity of labels on the expressiveness has been studied in prior work [12].

The second aspect, which is the focus of this paper, is the granularity of labeling (which is different from the granularity of the label itself). It pertains to the extent to which labeling is used on a program. Under this classification, a fine-grained type system is one which labels every program value individually by including a label in every type. For instance, FlowCaml [5] is a fine-grained IFC type system for ML. Every type constructor in FlowCaml is annotated with a security label. For example, $(A^H \times B^L)^L$ could be the type of low (public) pairs whose first component is high (secret) and second component is low. Here, H and L are standard labels used to denote high and low data respectively. For soundness, the label of a value must be an upper-bound on the labels of all the values that the value depends on. For instance, adding a low value to a high value must produce a high value. Therefore, a fine-grained type system must track such flows through all the operations in the language. These principles are embodied in several other type systems besides FlowCaml [1, 6–8].

Coarse-grained type systems [2, 9–11], on the other hand, are very different. They do not assign a label to every individual value. Instead, a label is associated with an entire sub-computation. Values in the scope of the sub-computation implicitly inherit the label of the sub-computation.[2] SLIO (the static fragment of HLIO [2]) is an instance of a coarse-grained IFC type system. It refines the type of a standard state monad with two labels ℓ_i and ℓ_o, as in $\mathbb{SLIO}\ \ell_i\ \ell_o\ \tau$.[3] The $\mathbb{SLIO}$ monad of SLIO is basically an instance of the Hoare state monad from [16] where ℓ_i represents the initial label of the computation (like the precondition) and ℓ_o represents the final label of the computation (like the postcondition). The label ℓ_i is an upper-bound on the level of the values that the computation received as input when it started. Hence, to prevent information leak, all the write effects of the computation must (conservatively) be above the level ℓ_i. Similarly, the label ℓ_o is an upper-bound on the level of values that the computation obtains from outside (by reading the heap) during its execution and, hence, the result of the computation must be assigned a label which is at least ℓ_o. Importantly, flows are tracked in a coarse-grained fashion; tracking is done only via the bind construct of the monad at sub-computation boundaries.

Looking at these vastly contrasting type systems for enforcing information flow control, a natural question that arises is that of their *relative* expressiveness [17]. By this we mean whether all programs that can be expressed in a fine-grained type system can also be expressed in a coarse-grained type system (and vice versa)? Upfront, it seems that a fine-grained type system should be able to express every program that a coarse-grained type system can, as the former abstracts less flows than the later. But the reverse (expressing a fine-grained type system in a coarse-grained one) is not immediately obvious. However, then one wonders if by structuring programs as extremely small sub-computations in a coarse-grained type system one may recover the expressiveness of a fine-grained type system after all.

In this paper, we show that both these intuitions are actually correct. We do this constructively, by showing an embedding of a fine-grained type system in a coarse-grained type system and vice versa. We do this in the setting of a higher-order language with state, to which we assign a fine-grained and a coarse-grained type system. For the fine-grained type system, we choose a system which is very close to SLAM [1] and the exception-free fragment of FlowCaml [5]. We call this type system FG here. For the coarse-grained type system, we work with SLIO*, a variant of SLIO mentioned above. We show that

[2] This idea traces lineage to IFC for operating systems [13–15], where the notion of sub-computation is a *process*, and a single label is associated with a process, but flows within a process are not tracked. However, this historic connection is not important for understanding the results of this paper.

[3] We use a different font face to represent the type constructor. This is done to differentiate the type constructor from the name of the type system.

FG and SLIO* are actually equi-expressive, by showing translations from FG to SLIO* and from SLIO* to FG.

While this is sufficient to prove the equi-expressiveness of fine-grained and coarse-grained IFC type systems, we note that the translation from FG to SLIO* is quite complex, making essential use of label quantification and constraint types. Going further, we examine the cause of this complexity. We show that this complexity arises due to the use of the Hoare state monad for enforcing IFC in SLIO*. To confirm this, we design a new coarse-grained type system, CG, which uses a monad different from that of SLIO*. CG's monad is also doubly indexed but does not interpret the two labels as precondition (input) and postcondition (output). This leads to a different proof theory for CG. A consequence of this change is that the translation from FG to CG is far simpler than that from FG to SLIO*. Our translations between FG and CG show not only the equi-expressiveness of FG and CG but also CG and SLIO* (indirectly through FG). Concretely, we present four translations in this paper: a) FG to SLIO* b) SLIO* to FG c) FG to CG and d) CG to FG. Taken separately, (a and b) or (c and d) suffices to show the equi-expressiveness of fine-grained and coarse-grained type systems. However, we use translations (b and c) and (d and a) to also show that SLIO* and CG are equi-expressive, i.e., the change from SLIO* to CG does not lead to any loss of expressiveness.

We believe that this resolves the question of the relative expressiveness of fine-grained and coarse-grained IFC type systems. A practical consequence of our theoretical results is that, since a coarse-grained type system involves less labeling annotations than a fine-grained one, a programmer can always choose to work with a coarse-grained type system without worrying about its expressiveness. In places where the precision of the fine-grained type system is really required, it can be *simulated* in the coarse-grained type system itself, following our (constructive) embedding of the fine-grained type system in the coarse-grained one.

As a second contribution of independent interest, we describe how to set up logical-relation based semantic models for the three type systems (FG, SLIO* and CG) over calculi with higher-order state. While models of IFC types have been studied before [1, 11, 18, 19], we do not know of any development that covers higher-order state. In fact, even in the absence of IFC, models of types with higher-order state are known to be difficult. We have the added complexity of dealing with information flow labels. Our models build on developments in the programming languages community in the past decade. Specifically, our models are based on step-indexed Kripke logical relations [20]. Like prior work on IFC types, 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 [21], the standard security property which says that public outputs of a program are not influenced by private inputs. This property is naturally defined using two runs. Using our models, we derive proofs of soundness of the fine-grained and both of 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. This implies that our translations preserve labels meaningfully [22]. Like all logical relations models, we expect that our models can be used for other purposes as well.

To summarize we make the following contributions through this work:

- We show the equi-expressiveness of fine-grained and coarse-grained IFC type systems using type- and semantics-preserving cross translations.

- We develop a new coarse-grained type IFC type system, CG, that breaks the input-output relation between the label indices of SLIO*'s state monad. This greatly simplifies the fine- to coarse-grained translation. At the same time, we show that CG is as expressive as SLIO*.
- We develop logical relations models for fine-grained and both of the coarse-grained type systems for a language with higher-order state.
- We give alternate proofs of soundness for FG and SLIO* using the logical relations models (the original proofs of the soundness of these type systems are syntactic in nature).

Accompanying appendix. Many technical details omitted from the paper to keep its length manageable are presented in an online appendix available from the authors' homepages.

Differences from the conference version. This paper is an extended version of a paper that appeared in CSF 2018 [23]. The conference version showed only the translations from FG to CG (and vice versa), omitting the translations from FG to SLIO* (and vice versa). This omitted material is covered in the completely new Section 3 of this paper. Accompanying this, Section 2.2 which recaps SLIO* and presents its novel logical relations model, is also new. Compared to CG, which is a type system we designed ourselves for the purpose of exposition, SLIO* is (very similar to) an existing type system and, hence, understanding its expressiveness relative to a fine-grained system is a more meaningful goal.

2. The type systems: FG and SLIO*

In this section, we describe the fine-grained type system and one of the two coarse-grained type systems we work with. Both these type systems are set up for higher-order stateful languages, but differ considerably in how they enforce IFC. The fine-grained type system, 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, SLIO*, 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 SLIO* 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 [1], but uses a presentation similar to Flow Caml, an IFC type system for ML [5]. 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 Fig. 1. The language has the usual constructs for functions, pairs, sums, and mutable references (heap locations). Along with that the language also includes constructs for label quantification ($\Lambda\ e$ is the introduction form and $e\ []$ is the elimination form) and constraint types ($\nu\ e$ is the introduction form and $e \bullet$ is the elimination form). 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

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

Type system: $\boxed{\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau}$

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

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

$$\frac{\Sigma; \Psi; \Gamma \vdash_{pc} e_1 : \tau_1 \qquad \Sigma; \Psi; \Gamma \vdash_{pc} e_2 : \tau_2}{\Sigma; \Psi; \Gamma \vdash_{pc} (e_1, e_2) : (\tau_1 \times \tau_2)^{\perp}}\ \text{FG-prod}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash_{pc} e : (\tau_1 \times \tau_2)^{\ell} \qquad \Sigma; \Psi \vdash \tau_1 \searrow \ell}{\Sigma; \Psi; \Gamma \vdash_{pc} \mathsf{fst}(e) : \tau_1}\ \text{FG-fst} \qquad \frac{\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau_1}{\Sigma; \Psi; \Gamma \vdash_{pc} \mathsf{inl}(e) : (\tau_1 + \tau_2)^{\perp}}\ \text{FG-inl}$$

$$\frac{\begin{array}{c}\Sigma; \Psi; \Gamma \vdash_{pc} e : (\tau_1 + \tau_2)^{\ell} \\ \Sigma; \Psi; \Gamma, x : \tau_1 \vdash_{pc \sqcup \ell} e_1 : \tau \qquad \Sigma; \Psi; \Gamma, y : \tau_2 \vdash_{pc \sqcup \ell} e_2 : \tau \qquad \Sigma; \Psi \vdash \tau \searrow \ell\end{array}}{\Sigma; \Psi; \Gamma \vdash_{pc} \mathsf{case}(e, x.e_1, y.e_2) : \tau}\ \text{FG-case}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash_{pc'} e : \tau' \qquad \Sigma; \Psi \vdash pc \sqsubseteq pc' \qquad \Sigma; \Psi \vdash \tau' <: \tau}{\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau}\ \text{FG-sub}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau \qquad \Sigma; \Psi \vdash \tau \searrow pc}{\Sigma; \Psi; \Gamma \vdash_{pc} \mathsf{new}\ e : (\mathsf{ref}\ \tau)^{\perp}}\ \text{FG-ref}$$

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

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

$$\frac{\Sigma, \alpha; \Psi; \Gamma \vdash_{\ell_e} e : \tau}{\Sigma; \Psi; \Gamma \vdash_{pc} \Lambda e : (\forall \alpha.(\ell_e, \tau))^{\perp}}\ \text{FG-FI} \qquad \frac{\Sigma; \Psi, c; \Gamma \vdash_{\ell_e} e : \tau}{\Sigma; \Psi; \Gamma \vdash_{pc} \nu\ e : (c \overset{\ell_e}{\Rightarrow} \tau)^{\perp}}\ \text{FG-CI}$$

Fig. 1. Language syntax and type system of FG (selected rules)

$$\frac{\Sigma;\Psi \vdash \ell \sqsubseteq \ell' \qquad \Sigma;\Psi \vdash \mathsf{A} <: \mathsf{A}'}{\Sigma;\Psi \vdash \mathsf{A}^{\ell} <: \mathsf{A}'^{\ell'}}\ \text{FGsub-label} \qquad \frac{}{\Sigma;\Psi \vdash \mathsf{b} <: \mathsf{b}}\ \text{FGsub-base}$$

$$\frac{}{\Sigma;\Psi \vdash \mathsf{ref}\ \tau <: \mathsf{ref}\ \tau}\ \text{FGsub-ref} \qquad \frac{\Sigma;\Psi \vdash \tau_1 <: \tau_1' \qquad \Sigma;\Psi \vdash \tau_2 <: \tau_2'}{\Sigma;\Psi \vdash \tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}\ \text{FGsub-prod}$$

$$\frac{\Sigma;\Psi \vdash \tau_1 <: \tau_1' \qquad \Sigma;\Psi \vdash \tau_2 <: \tau_2'}{\Sigma;\Psi \vdash \tau_1 + \tau_2 <: \tau_1' + \tau_2'}\ \text{FGsub-sum}$$

$$\frac{\Sigma;\Psi \vdash \tau_1' <: \tau_1 \qquad \Sigma;\Psi \vdash \tau_2 <: \tau_2' \qquad \Sigma;\Psi \vdash \ell_e' \sqsubseteq \ell_e}{\Sigma;\Psi \vdash \tau_1 \xrightarrow{\ell_e} \tau_2 <: \tau_1' \xrightarrow{\ell_e'} \tau_2'}\ \text{FGsub-arrow}$$

$$\frac{}{\Sigma;\Psi \vdash \mathsf{unit} <: \mathsf{unit}}\ \text{FGsub-unit} \qquad \frac{\Sigma,\alpha;\Psi \vdash \tau_1 <: \tau_2}{\Sigma;\Psi \vdash \forall\alpha.\tau_1 <: \forall\alpha.\tau_2}\ \text{FGsub-forall}$$

$$\frac{\Sigma;\Psi \vdash c_2 \implies c_1 \qquad \Sigma;\Psi, c_2 \vdash \tau_1 <: \tau_2}{\Sigma;\Psi \vdash c_1 \Rightarrow \tau_1 <: c_2 \Rightarrow \tau_2}\ \text{FGsub-constraint}$$

Fig. 2. FG subtyping

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 Fig. 1.

Typing rules. FG uses the typing judgment $\Sigma;\Psi;\Gamma \vdash_{pc} e : \tau$. As usual, Γ maps free variables of e to their types, Σ is a set of a label variables and Ψ is a set of label constraints. The judgment basically says that, an expression e is of type τ under the assumptions specified by Σ (a set of a label variables), Ψ (a set of label constraints) and Γ (a mapping from a variable to its type). 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 $\tau_1 \xrightarrow{\ell_e} \tau_2$, type for label quantification $\forall\alpha.(\ell_e, \tau)$ and constraint type $c \overset{\ell_e}{\Rightarrow} \tau$. Here, ℓ_e is a lower bound on the write effects of the body of the function.

FG's typing rules are shown in Fig. 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) states 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. FG-FE and FG-CE are similar.

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 (described in Fig. 2) 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 6.

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 new 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 [20] and is more directly based on a model of types in a different domain, namely, incremental computational complexity [24]. 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 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 [1, 11, 18] 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.

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

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

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

Fig. 3. Unary value, expression, and heap conformance relations for FG

Unary interpretation. The unary interpretation of types is shown in Fig. 3. 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 allocates more locations. A second important component used in the interpretation is a *step-index*, written m or n [25]. 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. Value relation for label quantification $(\forall \alpha.(\ell_e, \tau))$ and constraint type $(c \xRightarrow{\ell_e} \tau)$ follows intuition similar to the function type. The type $\mathsf{ref}\ \tau$ contains all locations a whose type

$$
\begin{aligned}
\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 \\
&\quad \{(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. \\
&\quad\quad ((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 \forall \alpha.(\ell_e,\tau) \rceil_V^{\mathcal{A}} &\triangleq \{(W,n,\Lambda e_1,\Lambda e_2) \mid \\
&\quad \forall W' \sqsupseteq W, n' < n, \ell' \in \mathcal{L}. \\
&\quad ((W',n',e_1,e_2) \in \lceil \tau[\ell'/\alpha] \rceil_E^{\mathcal{A}}) \wedge \\
&\quad \forall \theta_l \sqsupseteq W.\theta_1, j, \ell'' \in \mathcal{L}.((\theta_l,j,e_1) \in \lfloor \tau[\ell''/\alpha] \rfloor_E^{\ell_e[\ell''/\alpha]}) \wedge \\
&\quad \forall \theta_l \sqsupseteq W.\theta_2, j, \ell'' \in \mathcal{L}.((\theta_l,j,e_2) \in \lfloor \tau[\ell''/\alpha] \rfloor_E^{\ell_e[\ell''/\alpha]})\} \\
\lceil c \overset{\ell_e}{\Rightarrow} \tau \rceil_V^{\mathcal{A}} &\triangleq \{(W,n,\nu e_1,\nu e_2) \mid \\
&\quad \forall W' \sqsupseteq W, n' < n. \\
&\quad \mathcal{L} \models c \implies (W',n',e_1,e_2) \in \lceil \tau \rceil_E^{\mathcal{A}} \wedge \\
&\quad \forall \theta_l \sqsupseteq W.\theta_1, j.\mathcal{L} \models c \implies (\theta_l,j,e_1) \in \lfloor \tau \rfloor_E^{\ell_e} \wedge \\
&\quad \forall \theta_l \sqsupseteq W.\theta_2, j.\mathcal{L} \models c \implies (\theta_l,j,e_2) \in \lfloor \tau \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{aligned}
$$

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

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

Fig. 4. Binary value, expression and heap conformance relations for FG

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 Fig. 4. 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 parameterized 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 $\lceil\Gamma\rceil_V^{\mathcal{A}}$, 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.

$$\begin{aligned}\lfloor\Gamma\rfloor_V &\triangleq \{(\theta, n, \delta) \mid \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\delta) \wedge \forall x \in \mathtt{dom}(\Gamma).\\ &\qquad (\theta, n, \delta(x)) \in \lfloor\Gamma(x)\rfloor_V\}\\ \lceil\Gamma\rceil_V^{\mathcal{A}} &\triangleq \{(W, n, \gamma) \mid \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\gamma) \wedge \forall x \in \mathtt{dom}(\Gamma).\\ &\qquad (W, n, \pi_1(\gamma(x)), \pi_2(\gamma(x))) \in \lceil\Gamma(x)\rceil_V^{\mathcal{A}}\}\end{aligned}$$

Theorem 2.2 (Unary fundamental theorem)**.** *If* $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau$, $\mathcal{L} \models \Psi\ \sigma$ *and* $(\theta, n, \delta) \in \lfloor\Gamma\ \sigma\rfloor_V$ *then* $(\theta, n, e\ \delta) \in \lfloor\tau\ \sigma\rfloor_E^{pc}$.

Theorem 2.3 (Binary fundamental theorem)**.** *If* $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau$, $\mathcal{L} \models \Psi\ \sigma$ *and* $(W, n, \gamma) \in \lceil\Gamma\rceil_V^{\mathcal{A}}$, *then* $(W, n, e\ (\gamma{\downarrow_1}), e\ (\gamma{\downarrow_2})) \in \lceil\tau\ \sigma\rceil_E^{\mathcal{A}}$, *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, SLIO*

In this section, we describe our first coarse-grained type system, SLIO*. It is a minor variant of SLIO – the static fragment of the hybrid type system HLIO considered in [2]. SLIO* is defined for a call-by-value calculus.[4] We introduce some changes relative to SLIO*. First, SLIO* does not include the `label` construct of SLIO. This is because the `label` construct is the same as the standard monadic `ret` construct specialized to the Labeled type, with an additional label check. This label check is not required for noninterference (our soundness criterion) and is only needed for containment [26]. Hence, we omit it from SLIO*. Also, we introduce label quantification and constraints in SLIO* as in FG. Label

[4]The choice of call-by-value evaluation is made to match the evaluation approach used for FG. The original SLIO [2] is defined for a language with call-by-name evaluation.

$$\text{Expressions} \quad e ::= x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}(e, x.e, y.e) \mid \mathsf{new}\ e \mid\ !e \mid e := e \mid () \mid \mathsf{Lb}(e) \mid \mathsf{unlabel}(e) \mid \mathsf{toLabeled}(e) \mid \mathsf{ret}(e) \mid \mathsf{bind}(e, x.e) \mid \Lambda e \mid e\ [] \mid \nu\ e \mid e \bullet$$

$$\text{Types} \quad \tau ::= \mathsf{b} \mid \mathsf{unit} \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \mathsf{ref}\ \ell\ \tau \mid \mathsf{Labeled}\ \ell\ \tau \mid \mathbb{SLIO}\ \ell_1\ \ell_2\ \tau \mid \forall\alpha.(\ell_e, \tau) \mid c \overset{\ell_e}{\Rightarrow} \tau$$

Type system: $\boxed{\Sigma; \Psi; \Gamma \vdash e : \tau}$

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

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \tau}{\Sigma; \Psi; \Gamma \vdash \mathsf{Lb}(e) : \mathsf{Labeled}\ \ell\ \tau}\ \text{SLIO}^*\text{-label} \qquad \frac{\Sigma; \Psi; \Gamma \vdash e : \mathsf{Labeled}\ \ell\ \tau}{\Sigma; \Psi; \Gamma \vdash \mathsf{unlabel}(e) : \mathbb{SLIO}\ \ell_i\ (\ell_i \sqcup \ell)\ \tau}\ \text{SLIO}^*\text{-unlabel}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \mathbb{SLIO}\ \ell_i\ \ell_o\ \tau}{\Sigma; \Psi; \Gamma \vdash \mathsf{toLabeled}(e) : \mathbb{SLIO}\ \ell_i\ \ell_i\ (\mathsf{Labeled}\ \ell_o\ \tau)}\ \text{SLIO}^*\text{-toLabeled}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \tau}{\Sigma; \Psi; \Gamma \vdash \mathsf{ret}(e) : \mathbb{SLIO}\ \ell_i\ \ell_i\ \tau}\ \text{SLIO}^*\text{-ret}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e_1 : \mathbb{SLIO}\ \ell_i\ \ell\ \tau \qquad \Sigma; \Psi; \Gamma, x : \tau \vdash e_2 : \mathbb{SLIO}\ \ell\ \ell_o\ \tau'}{\Sigma; \Psi; \Gamma \vdash \mathsf{bind}(e_1, x.e_2) : \mathbb{SLIO}\ \ell_i\ \ell_o\ \tau'}\ \text{SLIO}^*\text{-bind}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \tau' \qquad \Sigma; \Psi \vdash \tau' <: \tau}{\Sigma; \Psi; \Gamma \vdash e : \tau}\ \text{SLIO}^*\text{-sub}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \mathsf{Labeled}\ \ell'\ \tau \qquad \Sigma; \Psi \vdash \ell \sqsubseteq \ell'}{\Sigma; \Psi; \Gamma \vdash \mathsf{new}\ e : \mathbb{SLIO}\ \ell\ \ell\ (\mathsf{ref}\ \ell'\ \tau)}\ \text{SLIO}^*\text{-ref}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \mathsf{ref}\ \ell\ \tau}{\Sigma; \Psi; \Gamma \vdash\ !e : \mathbb{SLIO}\ \ell'\ \ell'\ (\mathsf{Labeled}\ \ell\ \tau)}\ \text{SLIO}^*\text{-deref}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e_1 : \mathsf{ref}\ \ell'\ \tau \qquad \Sigma; \Psi; \Gamma \vdash e_2 : \mathsf{Labeled}\ \ell'\ \tau \qquad \Sigma; \Psi \vdash \ell \sqsubseteq \ell'}{\Sigma; \Psi; \Gamma \vdash e_1 := e_2 : \mathbb{SLIO}\ \ell\ \ell\ \mathsf{unit}}\ \text{SLIO}^*\text{-assign}$$

$$\frac{\Sigma, \alpha; \Psi; \Gamma \vdash e : \tau}{\Sigma; \Gamma \vdash \Lambda e : \forall\alpha.\tau}\ \text{SLIO}^*\text{-FI} \qquad \frac{\Sigma; \Psi; \Gamma \vdash e : \forall\alpha.\tau \qquad \text{FV}(\ell) \in \Sigma}{\Sigma; \Psi; \Gamma \vdash e\ [] : \tau[\ell/\alpha]}\ \text{SLIO}^*\text{-FE}$$

$$\frac{\Sigma; \Psi, c; \Gamma \vdash e : \tau}{\Sigma; \Gamma \vdash \nu\ e : c \Rightarrow \tau}\ \text{SLIO}^*\text{-CI} \qquad \frac{\Sigma; \Psi; \Gamma \vdash e : c \Rightarrow \tau \qquad \Sigma; \Psi \vdash c}{\Sigma; \Psi; \Gamma \vdash e \bullet : \tau}\ \text{SLIO}^*\text{-CE}$$

Fig. 5. Syntax and type system of SLIO* (selected rules)

quantification and constraints are required to make the translation from FG to SLIO* work, as we explain later.

$$\frac{\mathcal{L} \vdash \tau <: \tau' \qquad \mathcal{L} \vdash \ell \sqsubseteq \ell'}{\mathcal{L} \vdash \mathsf{Labeled}\ \ell\ \tau <: \mathsf{Labeled}\ \ell'\ \tau'}\ \text{SLIO}^*\text{sub-labeled}$$

$$\frac{\mathcal{L} \vdash \tau <: \tau' \qquad \mathcal{L} \vdash \ell_i' \sqsubseteq \ell_i \qquad \mathcal{L} \vdash \ell_o \sqsubseteq \ell_o'}{\mathcal{L} \vdash \mathbb{SLIO}\ \ell_i\ \ell_o\ \tau <: \mathbb{SLIO}\ \ell_i'\ \ell_o'\ \tau'}\ \text{SLIO}^*\text{sub-monad}$$

Fig. 6. SLIO* subtyping (selected rules)

Unlike FG, SLIO* makes a clear separation between pure and impure types. On the pure side, SLIO* includes types for an effect-free function, product, sum, label quantification and constraints. Both the intro and the elim forms for these types are completely free from IFC (label) checks. Additionally, SLIO* has a type for specifying labeled expressions. This type is written $\mathsf{Labeled}\ \ell\ \tau$, where ℓ specifies the confidentiality associated with the type τ. This represents the type of a labeled expression which is evaluated eagerly. References serve as sources and sinks for data and hence must have well-defined confidentiality associated with them. This is done by associating a label with the reference type, $\mathsf{ref}\ \ell\ \tau$. The label ℓ represents the label of the value *stored* in the reference or, in other words, a well-typed reference of type $\mathsf{ref}\ \ell\ \tau$ will always contain a value of type $\mathsf{Labeled}\ \ell\ \tau$. On the impure side, SLIO* represents effectful computations with a monadic type, i.e., all computations which can cause effects must be isolated in a monad. Hence, *pc* (the lower-bound on the write effect) is only relevant inside a monad. This is reflected by modeling (effectful) computations using a state monad (at the type level) which carries the initial and final *pc* as the state. The monadic type is represented by $\mathbb{SLIO}\ \ell_i\ \ell_o\ \tau$. ℓ_i and ℓ_o describe the initial and the final *pc*, and τ is the type of the value returned by forcing the computation of this monadic type.[5] It is an invariant of the SLIO* type system that if $e : \mathbb{SLIO}\ \ell_i\ \ell_o\ \tau$ then $\ell_i \sqsubseteq \ell_o$. As in the FG type system, ℓ_i is used as a lower-bound on the write-effects (as seen in the SLIO*-ref and SLIO*-assign rules explained later). In addition to that, SLIO* also has as an explicit representation for an upper-bound on the read effects (the final *pc* ℓ_o in the $\mathbb{SLIO}\ \ell_i\ \ell_o\ \tau$ type). This is in tandem with the core operational philosophy of coarse-grained IFC – the context label must be an upper bound on the confidentiality of the secrets read so far.

Having explained the various types of the SLIO* type system, we now describe some of its selected typing rules from Fig. 5 (note that we have omitted the type rules for pure expressions from Fig. 5 as they are completely standard and do not refer to labels at all). The typing judgment for the SLIO* type system is given by $\Sigma; \Psi; \Gamma \vdash e : \tau$ where Σ, Ψ and Γ represent the typing environment as in the FG type system. Notice that SLIO*'s typing judgment does not carry a global *pc* because effects in SLIO* are localized to the monads only and are not pervasive unlike the FG system. We start with SLIO*-ret (the intro form for the monadic type). $\mathsf{ret}(e)$ takes an expression of type τ and returns a computation of type $\mathbb{SLIO}\ \ell_i\ \ell_i\ \tau$ causing no change to the starting *pc* (which is ℓ_i). The elim form, $\mathsf{bind}(e_1, x.e_2)$, for the monadic type represents sequencing of two computations of types $\mathbb{SLIO}\ \ell\ \ell_1\ \tau$ and $\tau \to \mathbb{SLIO}\ \ell_1\ \ell'\ \tau'$ to obtain a computation of type $\mathbb{SLIO}\ \ell\ \ell'\ \tau'$. bind is the only locus of dependency tracking in this type system. This is reflected by typing the continuation (e_2) in a starting *pc* which equals the ending *pc* of the first computation (e_1). The SLIO*-label type rule says that if an expression e has type τ then the

[5] We use the term forcing to mean execution of the computation of the monadic type

expression $\mathsf{Lb}(e)$ has type $\mathsf{Labeled}\ \ell\ \tau$.[6] While $\mathsf{Lb}(e)$ represents a pure expression, its dual $\mathsf{unlabel}(e)$ represents a computation which when forced in a starting *pc* of ℓ_i will end up with a ending *pc* of $\ell_i \sqcup \ell$ where ℓ is the label on the type of e.

All the rules related to references perform heap-related effects and thus all of them have monadic types. SLIO*-ref and SLIO*-assign ensure that the reference created and modified respectively both get values whose confidentiality is above the confidentiality of the starting *pc* (lower-bound on the write effect denoted by ℓ in the type rules). Everything we have explained up to now models the conventional coarse-grained tracking, where reading (unlabeling) a secret value by a computation must make the *pc* secret for its continuation. This can be a problem when the continuation wants to produce a low output which does not depend on the read secret as this leads to a *label creep* – a situation where the output has a label higher than it ought to – and reduces precision. To help the programmer circumvent such label creep, SLIO* has a construct ($\mathsf{toLabeled}$) which can prevent the *pc* from being raised at the cost of returning a labeled value. $\mathsf{toLabeled}(e)$ coerces an expression (e) of type $\mathbb{SLIO}\ \ell_i\ \ell_o\ \tau$ to an expression of type $\mathbb{SLIO}\ \ell_i\ \ell_i\ (\mathsf{Labeled}\ \ell_o\ \tau)$. As a result the continuation of $\mathsf{toLabeled}(e)$ is not forced to start in an elevated *pc* of ℓ_o. Instead, it can start in ℓ_i and *only if* the continuation actually unlabels the result of the first expression (now of type $\mathsf{Labeled}\ \ell_o\ \tau$) will the *pc* be raised to ℓ_o.

Selected subtyping rules for SLIO* are described in Fig. 6. $\mathsf{Labeled}\ \ell\ \tau$ is co-variant in ℓ; and $\mathbb{SLIO}\ \ell_i\ \ell_o\ \tau$ is contra-variant in ℓ_i and co-variant in ℓ_o. This is natural as ℓ_i is a pre-condition and ℓ_o is a post-condition.

We prove soundness for SLIO* 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 SLIO*)**.** *Suppose (1)* $\ell_i \not\sqsubseteq \ell$*, (2)* $x : \mathsf{Labeled}\ \ell_i\ \tau \vdash e : \mathbb{SLIO}\ \_\ \ell\ \mathsf{bool}$*, 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 SLIO**

We build a new semantic model of SLIO*'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 SLIO*, all the constraints on heap updates move from the expression relations to the value relations at the monadic types as described in Fig. 7 and Fig. 8. Due to the similarity in the structure with FG's model, we defer the remaining details of the SLIO*'s model to the appendix.

3. Translations between FG and SLIO*

In this section, we describe our translations from FG to SLIO* and vice-versa, thus showing that these two type systems are equally expressive. We start with the translation from FG to SLIO*.

[6] Note that the Lb construct does not carry any runtime label. This is because all the enforcement is static in the type system so carrying runtime labels would be redundant.

$$
\begin{aligned}
\lfloor \mathbb{SLIO}\ \ell_1\ \ell_2\ \tau \rfloor_V \triangleq \{(\theta, m, e) \mid\ & \\
& \forall k \leqslant m, \theta_e \sqsupseteq \theta, H, j.(k, H) \triangleright \theta_e \wedge (H, v) \Downarrow_j^f (H', v') \wedge j < k \implies \\
& \exists \theta' \sqsupseteq \theta_e.(k-j, H') \triangleright \theta' \wedge (\theta', k-j, v') \in \lfloor \tau \rfloor_V \wedge \\
& (\forall a.H(a) \neq H'(a) \implies \exists \ell'.\theta_e(a) = \mathsf{Labeled}\ \ell'\ \tau' \wedge \ell_1 \sqsubseteq \ell') \wedge \\
& (\forall a \in \mathtt{dom}(\theta') \backslash \mathtt{dom}(\theta_e).\theta'(a) \searrow \ell_1)\}
\end{aligned}
$$

$$
\lfloor \tau \rfloor_E \triangleq \{(\theta, n, e) \mid \forall i < n.e \Downarrow_i v \implies (\theta, n-i, v) \in \lfloor \tau \rfloor_V\}
$$

Fig. 7. Unary logical relation for SLIO* (a snippet)

$$
\begin{aligned}
\lceil \mathbb{SLIO}\ \ell_1\ \ell_2\ \tau \rceil_V^{\mathcal{A}} \triangleq \{(W, n, v_1, v_2) \mid\ & \\
& \Big(\forall k \leqslant n, W_e \sqsupseteq W, H_1, H_2.(k, H_1, H_2) \triangleright W_e \wedge \\
& \forall v_1', v_2', j.(H_1, v_1) \Downarrow_j^f (H_1', v_1') \wedge (H_2, v_2) \Downarrow^f (H_2', v_2') \wedge j < k \implies \\
& \exists W' \sqsupseteq W_e.(k-j, H_1', H_2') \triangleright W' \wedge \mathit{ValEq}(\mathcal{A}, W', k-j, \ell_2, v_1', v_2', \tau)\Big) \wedge \\
& \forall l \in \{1,2\}.\Big(\forall k, \theta_e \sqsupseteq W.\theta_l, H, j.(k, H) \triangleright \theta_e \wedge (H, v_l) \Downarrow_j^f (H', v_l') \wedge j < k \implies \\
& \exists \theta' \sqsupseteq \theta_e.(k-j, H') \triangleright \theta' \wedge (\theta', k-j, v_l') \in \lfloor \tau \rfloor_V \wedge \\
& (\forall a.H(a) \neq H'(a) \implies \exists \ell'.\theta_e(a) = \mathsf{Labeled}\ \ell'\ \tau' \wedge \ell_1 \sqsubseteq \ell') \wedge \\
& (\forall a \in \mathtt{dom}(\theta') \backslash \mathtt{dom}(\theta_e).\theta'(a) \searrow \ell_1)\Big)\}
\end{aligned}
$$

$$
\lceil \tau \rceil_E^{\mathcal{A}} \triangleq \{(W, n, e_1, e_2) \mid \forall i < n.e_1 \Downarrow_i v_1 \wedge e_2 \Downarrow v_2 \implies (W, n-i, v_1, v_2) \in \lceil \tau \rceil_V^{\mathcal{A}}\}
$$

Fig. 8. Binary logical relation for SLIO* (a snippet)

*3.1. Translating FG to SLIO**

Translating FG to SLIO* is surprisingly very difficult. A translation was attempted previously in [17] but it only works for a subset of FG. The key idea behind our FG to SLIO* translation is to *index* the type translation function with a label α; the type translation takes the form $(\!|\tau|\!)_\alpha$. The index α can be thought of as a label propagated from the outer context, i.e., an additional label that protects τ. This indexing enables us to appropriately propagate labels in the translated SLIO* types.

We begin by looking at the translation of the typing judgment for FG, which is essentially the type preservation theorem that we would eventually like to prove. Here, $(\!|\Gamma|\!)_{\overline{\beta'}}$ is the pointwise lifting of the type translation to FG contexts ($\overline{\beta'}$ is a list of labels, of the same length as Γ).

Theorem 3.1. *Suppose $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau$ in FG. Then, there exists e' such that $\Sigma; \Psi; \Gamma \vdash_{pc} e : \tau \rightsquigarrow e'$ and for any $\alpha', \overline{\beta'}, \gamma'$ with $\overline{\beta'} \sqcup \gamma' \sqsubseteq pc \sqcap \alpha'$, there is a derivation of $\Sigma; \Psi; (\!|\Gamma|\!)_{\overline{\beta'}} \vdash e' : \mathbb{SLIO}\ \gamma'\ \gamma'\ (\!|\tau|\!)_{\alpha'}$ in SLIO*.*

Theorem 3.1 basically states that every well-typed FG expression e can be translated into some SLIO* expression e' s.t. $\Sigma; \Psi; (\!|\Gamma|\!)_{\overline{\beta'}} \vdash e' : \mathbb{SLIO}\ \gamma'\ \gamma'\ (\!|\tau|\!)_{\alpha'}$ for every $\alpha', \overline{\beta'}$, and γ' with $\overline{\beta'} \sqcup \gamma' \sqsubseteq pc \sqcap \alpha'$.

The constraint $\overline{\beta'} \sqcup \gamma' \sqsubseteq pc \sqcap \alpha'$ is basically a succinct representation of the following four constraints: $\overline{\beta'} \sqsubseteq pc, \gamma' \sqsubseteq pc, \overline{\beta'} \sqsubseteq \alpha'$ and $\gamma' \sqsubseteq \alpha'$. From the app rule of FG we know that the argument to the function can have a write effect of at least pc. We encode this using the $\overline{\beta'} \sqsubseteq pc$ constraint. Similarly we know that in FG, pc represents a lower bound on the write effect. This is encoded using the $\gamma' \sqsubseteq pc$ constraint. The other two constraints are best explained by appealing to their need in the type preservation proof. The constraint $\overline{\beta'} \sqsubseteq \alpha'$ is required to type check the variable case: A variable of type $(\!|\tau|\!)_{\beta_i}$ in the context can be type checked at $(\!|\tau|\!)_\alpha$ for some α s.t. $\beta_i \sqsubseteq \alpha$ (we prove this as a lemma in the appendix). The last constraint $\gamma' \sqsubseteq \alpha$ is required for the type-preservation of elim forms of type constructors.

We describe the full type translation function in Fig. 9. The type translation function is indexed with a label ℓ. The interesting case is the translation of the function type. The constraints in the translation of this type are similar to the ones we have in the theorem above. Additionally, we have two more constraints – $\ell \sqsubseteq \alpha$ and $\ell \sqsubseteq \ell_e$. Both the constraints are required for technical reasons that show up in the proof of type preservation of the introduction rule for function types.

$$
\begin{aligned}
(\!|\mathsf{b}|\!)_\ell &\triangleq \mathsf{b} \\
(\!|\mathsf{unit}|\!)_\ell &\triangleq \mathsf{unit} \\
(\!|\tau_1 \xrightarrow{\ell_e} \tau_2|\!)_\ell &\triangleq \forall \alpha, \beta, \gamma. (\ell \sqcup \beta \sqcup \gamma \sqsubseteq \alpha \sqcap \ell_e) \Rightarrow (\!|\tau_1|\!)_\beta \to \mathbb{SLIO}\ \gamma\ \gamma\ (\!|\tau_2|\!)_\alpha \\
(\!|\tau_1 \times \tau_2|\!)_\ell &\triangleq (\!|\tau_1|\!)_\ell \times (\!|\tau_2|\!)_\ell \\
(\!|\tau_1 + \tau_2|\!)_\ell &\triangleq (\!|\tau_1|\!)_\ell + (\!|\tau_2|\!)_\ell \\
(\!|\mathsf{ref}\ \mathsf{A}^{\ell'}|\!)_\ell &\triangleq \mathsf{ref}\ \ell'\ (\!|\mathsf{A}|\!)_{\ell'} \\
(\!|\forall \alpha.(\ell_e, \tau)|\!)_\ell &\triangleq \forall \alpha, \alpha', \gamma. (\ell \sqcup \gamma \sqsubseteq \alpha' \sqcap \ell_e) \Rightarrow \mathbb{SLIO}\ \gamma\ \gamma\ (\!|\tau|\!)_{\alpha'} \\
(\!|c \xRightarrow{\ell_e} \tau|\!)_\ell &\triangleq \forall \alpha, \gamma. (c \wedge \ell \sqcup \gamma \sqsubseteq \alpha \sqcap \ell_e) \Rightarrow \mathbb{SLIO}\ \gamma\ \gamma\ (\!|\tau|\!)_\alpha \\
\\
(\!|\mathsf{A}^{\ell'}|\!)_\ell &\triangleq \mathsf{Labeled}\ (\ell \sqcup \ell')\ (\!|\mathsf{A}|\!)_{\ell \sqcup \ell'}
\end{aligned}
$$

Fig. 9. Type translation from FG to SLIO*

Given this translation of types, we next define a type derivation-directed translation of expressions. This translation is formalized by the judgment $\Sigma; \Psi; \Gamma \vdash_{pc} e_s : \tau \rightsquigarrow e_t$. The judgment means that translating the source expression e_s, which has the typing derivation $\Sigma; \Psi; \Gamma \vdash_{pc} e_s$, 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 Fig. 10. The thing to keep in mind while reading the rules is that e_t should have the type $\mathbb{SLIO}\ \gamma'\ \gamma'\ (\!|\tau|\!)_{\alpha'}$ for any $\alpha', \overline{\beta'}, \gamma'$ s.t. $\overline{\beta'} \sqcup \gamma' \sqsubseteq pc \sqcap \alpha'$ (as stated in Theorem 3.1).

Selected rules for the expression translation from FG (source) to SLIO* (target) terms are described in Fig. 10. We illustrate how the translation works using one rule, FC-app. In this rule, inductively we obtain a translation of e_1, i.e., e_{c1} with the type $\mathbb{SLIO}\ \gamma'\ \gamma'\ (\!|(\tau_1 \xrightarrow{\ell_e} \tau_2)^\ell|\!)_{\beta' \sqcup \gamma'}$, where $\beta' = \bigcup_{\beta_i \in \overline{\beta'}} \beta_i$ which is equal to $\mathbb{SLIO}\ \gamma'\ \gamma'\ (\mathsf{Labeled}\ ((\beta' \sqcup \gamma') \sqcup \ell)\ (\forall \alpha, \beta, \gamma. (((\beta' \sqcup \gamma') \sqcup \ell) \sqcup \beta \sqcup \gamma \sqsubseteq \alpha \sqcap \ell_e) \Rightarrow (\!|\tau_1|\!)_\beta \to \mathbb{SLIO}\ \gamma\ \gamma\ (\!|\tau_2|\!)_\alpha))$. Similarly, the translation of e_2, i.e., e_{c2} has type $\mathbb{SLIO}\ \gamma'\ \gamma'\ (\!|\tau_1|\!)_{\beta' \cup \gamma'}$. We wish to construct a term of type $\mathbb{SLIO}\ \gamma'\ \gamma'\ (\!|\tau_2|\!)_{\beta' \cup \gamma'}$.

$$\frac{}{\Sigma;\Psi;\Gamma, x:\tau \vdash_{pc} x:\tau \rightsquigarrow \mathsf{ret}\ x}\ \text{FC-var}$$

$$\frac{\Sigma;\Psi;\Gamma, x:\tau_1 \vdash_{\ell_e} e:\tau_2 \rightsquigarrow e_{c1}}{\Sigma;\Psi;\Gamma \vdash_{pc} \lambda x.e : (\tau_1 \xrightarrow{\ell_e} \tau_2)^{\perp} \rightsquigarrow \mathsf{ret}(\mathsf{Lb}(\Lambda\Lambda\Lambda(\nu(\lambda x.e_{c1}))))}\ \text{FC-lam}$$

$$\frac{\begin{array}{c}\Sigma;\Psi;\Gamma \vdash_{pc} e_1 : (\tau_1 \xrightarrow{\ell_e} \tau_2)^{\ell} \rightsquigarrow e_{c1} \\ \Sigma;\Psi;\Gamma \vdash_{pc} e_2 : \tau_1 \rightsquigarrow e_{c2} \qquad \Sigma;\Psi \vdash \ell \sqcup pc \sqsubseteq \ell_e \qquad \Sigma;\Psi \vdash \tau \searrow \ell\end{array}}{\Sigma;\Psi;\Gamma \vdash_{pc} e_1\ e_2 : \tau_2 \rightsquigarrow \texttt{coerce\_taint}(\mathsf{bind}(e_{c1}, a.\mathsf{bind}(e_{c2}, b.\mathsf{bind}(\mathsf{unlabel}\ a, c.(c[][][]\bullet)\ b))))}\ \text{FC-app}$$

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

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

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

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

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

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

$\texttt{coerce\_taint} : \mathbb{SLIO}\ \gamma\ \alpha_c\ \tau' \to \mathbb{SLIO}\ \gamma\ \gamma\ \tau' \quad$ when $\tau' = \mathsf{Labeled}\ \alpha'_c\ \tau$ and $\Sigma, \Psi \models \alpha_c \sqsubseteq \alpha'_c$
$\texttt{coerce\_taint} \triangleq \lambda x.\mathsf{toLabeled}(\mathsf{bind}(x, y.\mathsf{unlabel}(y)))$

Fig. 10. Expression translation for FG to SLIO* (selected rules)

To do this, we bind e_{c1} to the variable a, which has the type $\mathsf{Labeled}\ ((\beta' \sqcup \gamma') \sqcup \ell)\ (\forall\alpha,\beta,\gamma.(((\beta' \sqcup \gamma') \sqcup \ell) \sqcup \beta \sqcup \gamma \sqsubseteq \alpha \sqcap \ell_e) \Rightarrow (\!|\tau_1|\!)_\beta \to \mathbb{SLIO}\ \gamma\ \gamma\ (\!|\tau_2|\!)_\alpha)$. Similarly, we bind e_{c2} to the variable b, which has the type $(\!|\tau_1|\!)_{\beta'\cup\gamma'}$. Next, we unlabel a and bind the result to variable c, which has the type $(\forall\alpha,\beta,\gamma.(((\beta' \sqcup \gamma') \sqcup \ell) \sqcup \beta \sqcup \gamma \sqsubseteq \alpha \sqcap \ell_e) \Rightarrow (\!|\tau_1|\!)_\beta \to \mathbb{SLIO}\ \gamma\ \gamma\ (\!|\tau_2|\!)_\alpha)$. However, due to the unlabeling, the *taint label on whatever computation we sequence after this bind must be at least* $\beta' \sqcup \gamma' \sqcup \ell$.

Next, we eliminate the quantifiers α with $\beta' \sqcup \gamma' \sqcup \ell$, β with $\beta' \sqcup \gamma'$ and γ with $\beta' \sqcup \gamma' \sqcup \ell$. This gives us an expression of the type $(((\beta' \sqcup \gamma') \sqcup \ell) \sqcup (\beta' \sqcup \gamma') \sqcup (\beta' \sqcup \gamma' \sqcup \ell) \sqsubseteq ((\beta' \sqcup \gamma') \sqcup \ell) \sqcap \ell_e) \Rightarrow (\!|\tau_1|\!)_{(\beta' \sqcup \gamma')} \rightarrow \mathbb{SLIO}\ (\beta' \sqcup \gamma' \sqcup \ell)\ (\beta' \sqcup \gamma' \sqcup \ell)\ (\!|\tau_2|\!)_{((\beta' \sqcup \gamma') \sqcup \ell)}$. The constraint to the left of $\Rightarrow$ is satisfiable as $\ell \sqcup pc \sqsubseteq \ell_e$ is given to us, so we eliminate the constraint type resulting in an expression $c[][][]\bullet$. Finally we apply $c[][][]\bullet$ to b to obtain the final result of type $\mathbb{SLIO}\ (\gamma')\ (\beta' \sqcup \gamma' \sqcup \ell)\ (\!|\tau_2|\!)_{((\beta' \sqcup \gamma') \sqcup \ell)}$ which is the same as $\mathbb{SLIO}\ (\gamma')\ (\beta' \sqcup \gamma' \sqcup \ell)\ (\mathsf{Labeled}\ (\ell_i \sqcup \beta' \sqcup \gamma' \sqcup \ell)\ (\!|\mathsf{A}|\!)_{(\ell_i \sqcup \beta' \sqcup \gamma' \sqcup \ell)})$, where $\tau_2 = A^{\ell_i}$. However, the desired type has the second index as γ' and subtyping can't be used to achieve that because of the co-variance of the monadic type on the second label. So, to achieve the desired type we use the `coerce_taint` function (described below) which has the type $\mathbb{SLIO}\ \gamma\ \alpha_c\ \tau' \rightarrow \mathbb{SLIO}\ \gamma\ \gamma\ \tau'$, when τ has the form $\tau' = \mathsf{Labeled}\ \alpha'_c\ \tau$ with $\Sigma, \Psi \models \alpha_c \sqsubseteq \alpha'_c$. Here, α_c is $(\beta' \sqcup \gamma' \sqcup \ell)$ and α'_c is $(\ell_i \sqcup \beta' \sqcup \gamma' \sqcup \ell)$. Since we are given that $\tau_2 \searrow \ell$, it follows that $\ell \sqsubseteq \ell_i$, and, hence, the required constraint is satisfied. This yields a term of type $\mathbb{SLIO}\ (\gamma')\ (\gamma')\ \mathsf{Labeled}\ (\ell_i \sqcup \beta' \sqcup \gamma' \sqcup \ell)\ (\!|\mathsf{A}|\!)_{(\ell_i \sqcup \beta' \sqcup \gamma' \sqcup \ell)}$, which is the same as $\mathbb{SLIO}\ (\gamma')\ (\gamma')\ (\!|\tau_2|\!)_{\beta' \cup \gamma'}$, as desired.

Importantly, the function `coerce_taint` uses toLabeled internally. The function is defined in Fig. 10. This pattern of using `coerce_taint`, which internally contains toLabeled, to restrict the taint 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.

3.1.1. Soundness of the translation

Our translation satisfies type preservation (Theorem 3.1). Type preservation is a necessary but insufficient condition for the goal we are after – showing equi-expressiveness of FG and SLIO*. In fact, one can obtain a *trivial* type-preserving translation from FG to SLIO* by simply translating every type to unit and every expression to $()$, but this translation would be useless as far as showing the equivalence of the two systems is concerned. Therefore, in this section we show that the translation from FG to SLIO* described above also preserves the semantics and the security of the source program. To that end, we build a cross-language model (based on Kripke step-indexed logical relations) for the translation from FG to SLIO*.

The cross-language relation (described in Fig. 11) relates a term of FG with a term of SLIO*. The *value relation* is defined by induction on the FG type and the step-index. It is a predicate over a tuple consisting of a world (denoted by ${}^s\theta$), which is a map from locations in the source language (FG) to their types, a step-index (denoted by m), a source value (of the FG type for which the relation is being defined) and a target value (of the corresponding translated type, obtained via the translation). Additionally, the relation is indexed by $\hat{\beta}$, a partial bijection describing the related locations between the source FG term and the target SLIO* term. The value relation for the arrow type, for instance, relates a FG λ-term to the corresponding translated term, if the given conditions are satisfied.

The *expression relation* embodies semantics preservation by stating that a FG's expression (e_s) is related to SLIO*'s expression (e_t) if an execution in FG (starting with some given heap) can be simulated by an execution in SLIO* (starting with a related heap) s.t. the resulting heaps and values are related by the heap well-formedness relation and the value relation, respectively. (We delegate the details of the heap well-formedness relation to the appendix.)

The logical relation can be extended to substitutions in a standard way. This enables us to state and prove the fundamental theorem.

Theorem 3.2 (Fundamental theorem)**.** $\forall \Sigma, \Psi, \Gamma, \tau, e_s, e_t, pc, \mathcal{L}, \delta^s, \delta^t, \sigma, {}^s\theta, n, \hat{\beta}.$

$$\Sigma; \Psi; \Gamma \vdash_{pc} e_s : \tau \rightsquigarrow e_t \wedge \mathcal{L} \models \Psi\ \sigma \wedge ({}^s\theta, n, \delta^s, \delta^t) \in \lfloor \Gamma\ \sigma \rfloor^{\hat{\beta}}_V \implies ({}^s\theta, n, e_s\ \delta^s, e_t\ \delta^t) \in \lfloor \tau\ \sigma \rfloor^{\hat{\beta}}_E$$

$$
\begin{aligned}
\lfloor \mathsf{b} \rfloor_V^{\hat{\beta}} &\triangleq \{({}^s\theta, m, {}^sv, {}^tv) \mid {}^sv \in [\![\mathsf{b}]\!] \wedge {}^tv \in [\![\mathsf{b}]\!] \wedge {}^sv = {}^tv\} \\
\lfloor \mathsf{unit} \rfloor_V^{\hat{\beta}} &\triangleq \{({}^s\theta, m, {}^sv, {}^tv) \mid {}^sv \in [\![\mathsf{unit}]\!] \wedge {}^tv \in [\![\mathsf{unit}]\!]\} \\
\lfloor \tau_1 \times \tau_2 \rfloor_V^{\hat{\beta}} &\triangleq \{({}^s\theta, m, ({}^sv_1, {}^sv_2), ({}^tv_1, {}^tv_2)) \mid \\
&\qquad ({}^s\theta, m, {}^sv_1, {}^tv_1) \in \lfloor \tau_1 \rfloor_V^{\hat{\beta}} \wedge ({}^s\theta, m, {}^sv_2, {}^tv_2) \in \lfloor \tau_2 \rfloor_V^{\hat{\beta}}\} \\
\lfloor \tau_1 + \tau_2 \rfloor_V^{\hat{\beta}} &\triangleq \{({}^s\theta, m, \mathsf{inl}\ {}^sv, \mathsf{inl}\ {}^tv) \mid ({}^s\theta, m, {}^sv, {}^tv) \in \lfloor \tau_1 \rfloor_V^{\hat{\beta}}\} \cup \\
&\qquad \{({}^s\theta, m, \mathsf{inr}\ {}^sv, \mathsf{inr}\ {}^tv) \mid ({}^s\theta, m, {}^sv, {}^tv) \in \lfloor \tau_2 \rfloor_V^{\hat{\beta}}\} \\
\lfloor \tau_1 \xrightarrow{\ell_e} \tau_2 \rfloor_V^{\hat{\beta}} &\triangleq \{({}^s\theta, m, \lambda x.e_s, \Lambda\Lambda(v(\lambda x.e_t))) \mid \\
&\qquad \forall {}^s\theta' \sqsupseteq {}^s\theta, {}^sv, {}^tv, j < m, \hat{\beta} \sqsubseteq \hat{\beta}'.({}^s\theta', j, {}^sv, {}^tv) \in \lfloor \tau_1 \rfloor_V^{\hat{\beta}'} \implies \\
&\qquad ({}^s\theta', j, e_s[{}^sv/x], e_t[{}^tv/x]) \in \lfloor \tau_2 \rfloor_E^{\hat{\beta}'}\} \\
\lfloor \forall \alpha.(\ell_e, \tau) \rfloor_V^{\hat{\beta}} &\triangleq \{({}^s\theta, m, \Lambda e_s, \Lambda\Lambda(v(e_t))) \mid \\
&\qquad \forall {}^s\theta' \sqsupseteq {}^s\theta, j < m, \ell' \in \mathcal{L}, \hat{\beta} \sqsubseteq \hat{\beta}'.({}^s\theta', j, e_s, e_t) \in \lfloor \tau[\ell'/\alpha] \rfloor_E^{\hat{\beta}'}\} \\
\lfloor c \xRightarrow{\ell_e} \tau \rfloor_V^{\hat{\beta}} &\triangleq \{({}^s\theta, m, ve_s, \Lambda(v(e_t))) \mid \\
&\qquad \mathcal{L} \models c \implies \forall {}^s\theta' \sqsupseteq {}^s\theta, j < m, \hat{\beta} \sqsubseteq \hat{\beta}'.({}^s\theta', j, e_s, e_t) \in \lfloor \tau \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 ({}^sa, {}^ta) \in \hat{\beta}\} \\
\lfloor \mathsf{A}^{\ell'} \rfloor_V^{\hat{\beta}} &\triangleq \{({}^s\theta, m, {}^sv, \mathsf{Lb}({}^tv)) \mid ({}^s\theta, m, {}^sv, {}^tv) \in \lfloor \mathsf{A} \rfloor_V^{\hat{\beta}}\}
\end{aligned}
$$

$$
\begin{aligned}
\lfloor \tau \rfloor_E^{\hat{\beta}} \triangleq \{&({}^s\theta, n, e_s, e_t) \mid \\
&\forall H_s, H_t.(n, H_s, H_t) \overset{\hat{\beta}}{\triangleright} {}^s\theta \wedge \forall i < n, {}^sv.(H_s, e_s) \Downarrow_i (H'_s, {}^sv) \implies \\
&\exists H'_t, {}^tv.(H_t, e_t) \Downarrow^f (H'_t, {}^tv) \wedge \exists {}^s\theta' \sqsupseteq {}^s\theta, \hat{\beta}' \sqsupseteq \hat{\beta}.(n - i, H'_s, H'_t) \overset{\hat{\beta}'}{\triangleright} {}^s\theta' \\
&\wedge ({}^s\theta', n - i, {}^sv, {}^tv) \in \lfloor \tau \rfloor_V^{\hat{\beta}'}\}
\end{aligned}
$$

$$
\begin{aligned}
(n, H_s, H_t) \overset{\hat{\beta}}{\triangleright} {}^s\theta \triangleq\ & \mathtt{dom}({}^s\theta) \subseteq \mathtt{dom}(H_s) \wedge \\
& \hat{\beta} \subseteq (\mathtt{dom}({}^s\theta) \times \mathtt{dom}(H_t)) \wedge \\
& \forall (a_1, a_2) \in \hat{\beta}.({}^s\theta, n - 1, H_s(a_1), H_t(a_2)) \in \lfloor {}^s\theta(a_1) \rfloor_V^{\hat{\beta}}
\end{aligned}
$$

$$
\begin{aligned}
\lfloor \Gamma \rfloor_V^{\hat{\beta}} \triangleq\ & \{({}^s\theta, n, \delta^s, \delta^t) \mid \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\delta^s) \wedge \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\delta^t) \wedge \\
& \forall x \in \mathtt{dom}(\Gamma).({}^s\theta, n, \delta^s(x), \delta^t(x)) \in \lfloor \Gamma(x) \rfloor_V^{\hat{\beta}}\}
\end{aligned}
$$

Fig. 11. Cross language value and expression relation for FG to SLIO*

This theorem immediately implies that the FG to SLIO* translation preserves semantics of terms. Moreover, combining this theorem with Theorem 3.1, we can re-derive the noninterference theorem of FG from that of SLIO*. This means that the translation preserves the intents of security labels or, in other words, it is security-preserving.

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \tau \rightsquigarrow e_F}{\Sigma; \Psi; \Gamma \vdash \mathsf{Lb}_\ell(e) : \mathsf{Labeled}\ \ell\ \tau \rightsquigarrow \mathsf{inl}(e_F)}\ \text{label}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \mathsf{Labeled}\ \ell\ \tau \rightsquigarrow e_F}{\Sigma; \Psi; \Gamma \vdash \mathsf{unlabel}(e) : \mathbb{SLIO}\ \ell_i\ (\ell_i \sqcup \ell)\ \tau \rightsquigarrow \lambda\_.e_F}\ \text{unlabel}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \mathbb{SLIO}\ \ell_i\ \ell_o\ \tau \rightsquigarrow e_F}{\Sigma; \Psi; \Gamma \vdash \mathsf{toLabeled}(e) : \mathbb{SLIO}\ \ell_i\ \ell_i\ (\mathsf{Labeled}\ \ell_o\ \tau) \rightsquigarrow \lambda\_.\mathsf{inl}(e_F\ ())}\ \text{toLabeled}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \tau \rightsquigarrow e_F}{\Sigma; \Psi; \Gamma \vdash \mathsf{ret}(e) : \mathbb{SLIO}\ \ell_i\ \ell_i\ \tau \rightsquigarrow \lambda\_.\mathsf{inl}(e_F)}\ \text{ret}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e_1 : \mathbb{SLIO}\ \ell_i\ \ell\ \tau \rightsquigarrow e_{F1} \qquad \Sigma; \Psi; \Gamma, x : \tau \vdash e_2 : \mathbb{SLIO}\ \ell\ \ell_o\ \tau' \rightsquigarrow e_{F2}}{\Sigma; \Psi; \Gamma \vdash \mathsf{bind}(e_1, x.e_2) : \mathbb{SLIO}\ \ell_i\ \ell_o\ \tau' \rightsquigarrow \lambda\_.\mathsf{case}(e_{F1}(), x.e_{F2}(), y.\mathsf{inr}())}\ \text{bind}$$

Fig. 12. Expression translation SLIO* to FG (selected rules only)

3.2. *Translating SLIO* to FG*

This section describes the translation in the other direction—from SLIO* to FG. The overall structure of this translation is similar to that of the earlier FG to SLIO* translation, given by a translation of types and a type-derivation-directed translation of terms. However, this translation is much simpler in the details, in particular, the type translation now does not require indexing with a label. Note that the superscript or subscript *s* (source) now marks elements of SLIO* and *t* (target) marks elements of FG.

The key idea of the translation is to map a source (SLIO*) 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 SLIO*, 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.

$$[\![\mathsf{b}]\!] \triangleq \mathsf{b}^\bot \qquad [\![\mathsf{ref}\ \ell\ \tau]\!] \triangleq (\mathsf{ref}\ ([\![\tau]\!] + \mathsf{unit})^\ell)^\bot$$
$$[\![\tau_1 \to \tau_2]\!] \triangleq ([\![\tau_1]\!] \xrightarrow{\top} [\![\tau_2]\!])^\bot \qquad [\![\mathbb{SLIO}\ \ell_1\ \ell_2\ \tau]\!] \triangleq (\mathsf{unit} \xrightarrow{\ell_1} ([\![\tau]\!] + \mathsf{unit})^{\ell_2})^\bot$$
$$[\![\tau_1 \times \tau_2]\!] \triangleq ([\![\tau_1]\!] \times [\![\tau_2]\!])^\bot \qquad [\![\mathsf{Labeled}\ \ell\ \tau]\!] \triangleq ([\![\tau]\!] + \mathsf{unit})^\ell$$
$$[\![\tau_1 + \tau_2]\!] \triangleq ([\![\tau_1]\!] + [\![\tau_2]\!])^\bot$$

The most interesting case of the translation is that for $\mathbb{SLIO}\ \ell_1\ \ell_2\ \tau$. Since a SLIO* 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 $(\llbracket\tau\rrbracket + \mathsf{unit})^{\ell_2}$. For example, $(\llbracket\tau\rrbracket \times \mathsf{unit})^{\ell_2}$ works equally well.

The expression translation is directed by source typing derivations and is defined by the judgment $\Sigma;\Psi;\Gamma \vdash e_s : \tau \rightsquigarrow e_t$, some of whose rules are shown in Fig. 12. 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 $(\llbracket\tau\rrbracket + \mathsf{unit})^{\ell}$ needs to be constructed.

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

Theorem 3.3 (Typing preservation)**.** *If $\Sigma;\Psi;\Gamma \vdash e_s : \tau$ in SLIO*, then there is a unique e_t such that $\Sigma;\Psi;\Gamma \vdash e_s : \tau \rightsquigarrow e_t$ and that e_t satisfies $\Sigma;\Psi;\llbracket\Gamma\rrbracket \vdash_\top e_t : \llbracket\tau\rrbracket$ in FG.*

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

We further prove that the translation preserves the semantics of programs. Our approach is the same as that for the FG to SLIO* translation—we set up a cross-language logical relation, this time indexed by SLIO* 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 SLIO* 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. CG: A new coarse-grained type system

We explained in Section 2.2 that SLIO*'s monadic type $\mathbb{SLIO}\ \ell_i\ \ell_o\ \tau$ is really a Hoare monad. The two labels ℓ_i and ℓ_o on the monadic type serve as input and output labels. For any well-typed program, they always satisfy the invariant that $\ell_i \sqsubseteq \ell_o$. However, as it turns out, this constraint is really not necessary for the soundness of SLIO*'s type system. Accordingly, in this section, we develop a similar type system CG, but which does not force the $\ell_i \sqsubseteq \ell_o$ restriction. CG interprets ℓ_i like a conventional *pc* and ℓ_o as a label on the output of the result (the two do not have to be related, exactly as in FG). It further turns out that removing the restriction $\ell_i \sqsubseteq \ell_o$ really simplifies the translation from FG, by allowing more flexibility in the labels in the target. As we show later, a translation from FG to CG can be done without requiring any label quantification and constraints unlike the translation from FG to SLIO* in Section 3.1. This suggests that a large part of the complexity in the translation of Section 3.1 in fact comes from the "Hoare" interpretation of the two labels on SLIO*'s monad.

CG is defined for the same language as SLIO*, but it has a different type system. Specifically, CG differs from SLIO* only in the rules for the monadic types (the impure part of the type system). These new rules are described in Fig. 13. CG's monadic type is written $\mathbb{C}\ \ell_i\ \ell_o\ \tau$. As mentioned above, in CG, the first label ℓ_i on the monadic type is like FG's *pc* label: It is a lower bound on the write effects of the computation ascribed by the monadic type. The second label ℓ_o can be thought of as the security label of the result and it is an upper bound on the read effects of the computation. No specific ordering relation is required or forced between the two labels.

CG's typing judgment is written $\Sigma;\Psi;\Gamma \vdash e : \tau$. Like SLIO*, there is no need for a *pc* on the judgment since effects are confined to the monad. 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

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

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

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \tau}{\Sigma; \Psi; \Gamma \vdash \mathsf{ret}(e) : \mathbb{C}\ \top\ \bot\ \tau}\ \text{CG-ret} \qquad \frac{\Sigma; \Psi; \Gamma \vdash e : \tau' \qquad \Sigma; \Psi \vdash \tau' <: \tau}{\Sigma; \Psi; \Gamma \vdash e : \tau}\ \text{CG-sub}$$

$$\frac{\Sigma; \Psi; \Gamma \vdash e : \tau}{\Sigma; \Psi; \Gamma \vdash \mathsf{Lb}(e) : \mathsf{Labeled}\ \ell\ \tau}\ \text{CG-label} \qquad \frac{\Sigma; \Psi; \Gamma \vdash e : \mathsf{Labeled}\ \ell\ \tau}{\Sigma; \Psi; \Gamma \vdash \mathsf{unlabel}(e) : \mathbb{C}\ \top\ \ell\ \tau}\ \text{CG-unlabel}$$

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

$$\frac{\Sigma; \Psi; \Gamma \vdash e_1 : \mathsf{ref}\ \ell\ \tau \qquad \Sigma; \Psi; \Gamma \vdash e_2 : \mathsf{Labeled}\ \ell\ \tau}{\Sigma; \Psi; \Gamma \vdash e_1 := e_2 : \mathbb{C}\ \ell\ \bot\ \mathsf{unit}}\ \text{CG-assign}$$

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

Fig. 13. CG's type system (selected rules)

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 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 $\mathbb{C}\ \ell\ \bot$ unit. The pc-label ℓ indicates a write effect at level ℓ.

The last typing rule we highlight pertains to the construct toLabeled(e). This construct transforms e of monadic type $\mathbb{C}\ \ell\ \ell'\ \tau$ to the type $\mathbb{C}\ \ell\ \bot$ (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 over-tainting subsequent computations. We make extensive use of this construct in our translation from FG to CG.

We prove soundness for CG by showing that every well-typed expression satisfies noninterference (Theorem 4.1). The proof is done by developing logical relations of the kind we saw earlier for FG and SLIO*. The logical relations for CG are exactly the same as that for SLIO*, so we don't describe them again.

Theorem 4.1 (Noninterference for CG)**.** *Suppose (1) $\ell_i \not\sqsubseteq \ell$, (2) x : Labeled $\ell_i\ \tau \vdash e : \mathbb{C}\ \_\ \ell$ bool, and (3) v_1, v_2 : 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*).*

5. Translations between FG and CG

In this section, we describe our translations from FG to CG and vice-versa. The purpose of describing these translations is two fold: 1) To show that these two type systems are equally expressive and, more importantly, to show that the translation from a fine-grained to a coarse-grained IFC type system does not have to be as complex as we saw in Section 3.1. We show the latter by describing a much simpler translation from FG to CG. 2) To show that CG and SLIO* are also equi-expressive despite the differences in their proof theories. The translation from CG to FG is very similar to the translation from SLIO* to FG. There are only minor differences in the proofs due to differences in the type system of the source. We omit this translation here as it adds no insight to what we already described in Section 3.2. Instead, we only describe the translation from FG to CG, which is substantially different from (and considerably simpler than) the translation from FG to SLIO*.

5.1. 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 in a very simple manner. This translation is directed by the type derivations in FG and preserves typing, semantics and security annotations. As before, we use the subscript or superscript s to indicate source (FG) elements, and t to indicate target (CG) elements.

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.

$$
\begin{aligned}
(\!|\mathsf{b}|\!) &\triangleq \mathsf{b} \\
(\!|\mathsf{unit}|\!) &\triangleq \mathsf{unit} \\
(\!|\tau_1 \xrightarrow{\ell_e} \tau_2|\!) &\triangleq (\!|\tau_1|\!) \to \mathbb{C}\ \ell_e\ \bot\ (\!|\tau_2|\!) \\
(\!|\forall\alpha.(\ell_e, \tau)|\!) &\triangleq \forall\alpha.\mathbb{C}\ \ell_e\ \bot\ (\!|\tau|\!) \\
(\!|c \overset{\ell_e}{\Rightarrow} \tau)|\!) &\triangleq c \Rightarrow \mathbb{C}\ \ell_e\ \bot\ (\!|\tau|\!)
\end{aligned}
\qquad
\begin{aligned}
(\!|\tau_1 \times \tau_2|\!) &\triangleq (\!|\tau_1|\!) \times (\!|\tau_2|\!) \\
(\!|\tau_1 + \tau_2|\!) &\triangleq (\!|\tau_1|\!) + (\!|\tau_2|\!) \\
(\!|\mathsf{ref}\ \tau|\!) &\triangleq \mathsf{ref}\ \ell\ (\!|\mathsf{A}|\!) \quad \text{where } \tau = \mathsf{A}^\ell \\
(\!|\mathsf{A}^\ell|\!) &\triangleq \mathsf{Labeled}\ \ell\ (\!|\mathsf{A}|\!)
\end{aligned}
$$

The translation should be self-explanatory. The only nontrivial 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 $(\!|\tau_2|\!)$. It is instructive to contrast this type translation to the type translation from FG to SLIO*. Now the translation of the function type does not need label quantification and constraints.

The expression translation uses ideas similar to that of the translation from FG to SLIO*, but the translated terms are simpler. We don't explain the term translation again, and only list some selected rules in Fig. 14.

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. Notice the simplicity of this theorem in contrast with the type preservation for the FG to SLIO* translation (Theorem 3.1).

Theorem 5.1 (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 non-interfering target programs (since target typing implies noninterference in the target).

Next, just like our previous translations we also show that this translation is semantics preserving by developing a cross-language relation between FG and CG terms. The cross-language relation is similar to (but not the same as) the cross-language relation we saw for the FG to SLIO* translation. We defer its description to the appendix. As before, we also show that we are able to re-derive FG's noninterference theorem from CG's noninterference theorem and properties of the translation.

6. 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 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

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

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

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

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

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

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

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

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

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

Fig. 14. Expression translation FG to CG (selected rules only)

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.

Full abstraction. Since our translations preserve typed-ness, they map well-typed source programs to non-interfering 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.

7. 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 [18]. 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 [1, 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 FlowCaml [5], 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 [6], 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 [19]. 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 [20], 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 [24]. 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 models for SLIO* and CG have no direct predecessor; we developed them 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.* [22] 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 [27] 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 non-interfering target programs, where the target noninterference is defined in a computational sense. Algehed and Russo [28] 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.

8. 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. We show this via cross-translations between a fine-grained type system, FG, and a coarse-grained type systems SLIO*. The translation from FG to SLIO* is quite complex due to the Hoare state monad implicitly used in SLIO*. We come up with a new coarse-grained type system CG which is not only equi-expressive with SLIO*, but also affords a much simpler translation from FG. A more foundational contribution of our work is a better understanding of semantic models of information flow types. To this end, we presented logical relations models of types for the fine-grained and both of the coarse-grained type systems presented in the paper.

Acknowledgments

This work was partly supported by the German Science Foundation (DFG) through the Collaborative Research Center "Methods and Tools for Understanding and Controlling Privacy" (SFB 1223).

References

[1] 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.
[2] 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.
[3] 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.
[4] T.H. Austin and C. Flanagan, Permissive dynamic information flow analysis, in: *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, (PLAS)*, 2010.
[5] F. Pottier and V. Simonet, Information Flow Inference for ML, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **25**(1) (2003).
[6] D.M. Volpano, C.E. Irvine and G. Smith, A Sound Type System for Secure Flow Analysis, *Journal of Computer Security (JCS)* **4**(2–3) (1996).

[7] A.C. Myers and B. Liskov, Protecting Privacy Using the Decentralized Label Model, *ACM Transactions on Software Engineering and Methodology (TOSEM)* **9**(4) (2000).
[8] 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.
[9] A.A. Matos and G. Boudol, On declassification and the non-disclosure policy, *Journal of Computer Security (JCS)* **17**(5) (2009).
[10] G. Boudol, Secure Information Flow as a Safety Property, in: *International Workshop on Formal Aspects in Security and Trust (FAST)*, 2008.
[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] 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.
[13] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek and R. Morris, Labels and Event Processes in the Asbestos Operating System, in: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005.
[14] N. Zeldovich, S. Boyd-Wickizer, E. Kohler and D. Mazières, Making Information Flow Explicit in HiStar, in: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
[15] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M.F. Kaashoek, E. Kohler and R. Morris, Information Flow Control for Standard OS Abstractions, in: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.
[16] A. Nanevski, G. Morrisett and L. Birkedal, Polymorphism and Separation in Hoare Type Theory, in: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2006.
[17] V. Rajani, I. Bastys, W. Rafnsson and D. Garg, Type systems for information flow control: The question of granularity, *SIGLOG News* **4**(1) (2017).
[18] 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.
[19] H. Mantel, D. Sands and H. Sudbrock, Assumptions and Guarantees for Compositional Noninterference, in: *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, 2011.
[20] 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.
[21] J.A. Goguen and J. Meseguer, Security policies and security models, in: *Proceedings of the IEEE Symposium on Security and Privacy*, 1982.
[22] G. Barthe, T. Rezk and A. Basu, Security types preserving compilation, *Computer Languages, Systems & Structures (CLSS)* **33**(2) (2007).
[23] V. Rajani and D. Garg, Types for Information Flow Control: Labeling Granularity and Semantic Models, in: *Proc. of the 31st IEEE Computer Security Foundations Symposium (CSF)*, 2018.
[24] 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.
[25] 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.
[26] D. Stefan, A. Russo, J.C. Mitchell and D. Mazières, Flexible Dynamic Information Flow Control in Haskell, in: *Proceedings of the ACM Symposium on Haskell (Haskell)*, 2011.
[27] 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.
[28] 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.