From Fine- to Coarse-Grained Dynamic Information Flow Control and Back

MARCO VASSENA, Chalmers University of Technology, Sweden
ALEJANDRO RUSSO, Chalmers University of Technology, Sweden
DEEPAK GARG, Max Planck Institute for Software Systems, Germany
VINEET RAJANI, Max Planck Institute for Software Systems, Germany
DEIAN STEFAN, University of California San Diego, USA

We show that fine-grained and coarse-grained dynamic information-flow control (IFC) systems are equally expressive. To this end, we mechanize two mostly standard languages, one with a fine-grained dynamic IFC system and the other with a coarse-grained dynamic IFC system, and prove a semantics-preserving translation from each language to the other. In addition, we derive the standard security property of non-interference of each language from that of the other, via our verified translation. This result addresses a longstanding open problem in IFC: whether coarse-grained dynamic IFC techniques are less expressive than fine-grained dynamic IFC techniques (they are not!). The translations also stand to have important implications on the usability of IFC approaches. The coarse- to fine-grained direction can be used to remove the label annotation burden that fine-grained systems impose on developers, while the fine- to coarse-grained translation shows that coarse-grained systems—which are easier to design and implement—can track information as precisely as fine-grained systems and provides an algorithm for automatically retrofitting legacy applications to run on existing coarse-grained systems.

Additional Key Words and Phrases: Information-flow control, verified source-to-source transformations, Agda

1 INTRODUCTION

Dynamic *information-flow control* (IFC) is a principled approach to protecting the confidentiality and integrity of data in software systems. Conceptually, dynamic IFC systems are very simple—they associate *security* levels or *labels* with every bit of data in the system to subsequently track and restrict the flow of labeled data throughout the system, e.g., to enforce a security property such as *non-interference* [Goguen and Meseguer 1982]. In practice, dynamic IFC implementations are considerably more complex, where the *granularity* of the tracking system alone has important implications for the usage of IFC technology in practice.Indeed, until somewhat recently [Roy et al. 2009; Stefan et al. 2017], granularity was the main distinguishing factor between dynamic IFC operating systems and programming languages. Most IFC operating systems (e.g., [Efstathopoulos et al. 2005; Krohn et al. 2007b; Zeldovich et al. 2006]) are *coarse-grained*, i.e., they track and enforce information flow at the granularity of a process or thread. Conversely, most programming languages with dynamic IFC (e.g., [Austin and Flanagan 2009; Hedin et al. 2014; Hritcu et al. 2013a; Yang et al. 2012; Zdancewic 2002]) track the flow of information in a more *fine-grained* fashion, e.g., at the granularity of program variables and references.

Authors' addresses: Marco Vassena, Chalmers University of Technology, Sweden; Alejandro Russo, Chalmers University of Technology, Sweden; Deepak Garg, Max Planck Institute for Software Systems, Germany; Vineet Rajani, Max Planck Institute for Software Systems, Germany; Deian Stefan, University of California San Diego, USA.

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

Dynamic coarse-grained IFC systems in the style of LIO [Buiras et al. 2015; Heule et al. 2015; Stefan et al. 2012, 2017, 2011; Vassena et al. 2017] have several advantages over dynamic fine-grained IFC systems. Such coarse-grained systems are often easier to design and implement—they inherently track less information. For example, LIO protects against control-flow-based *implicit flows* by tracking information at a coarse-grained level—to branch on secrets, LIO programs must first taint the context where secrets are going to be observed. Finally, coarse-grained systems often require considerably fewer programmer annotations—unlike fine-grained ones. More specifically, developers often only need to annotate a single-label to protect everything in the scope of a thread or process responsible to handle sensitive data.

Unfortunately, these advantages of coarse-grained systems give up on the many benefits of fine-grained ones. For instance, one main drawback of coarse-grained systems is that it requires developers to compartmentalize their application in order to avoid both false alarms and the *label creep* problem, i.e., wherein the program gets too "tainted" to do anything useful. To this end, fine-grained systems often create special abstractions (e.g., event processes [Efstathopoulos et al. 2005], gates [Zeldovich et al. 2006], and security regions [Roy et al. 2009]) that compensate for the conservative approximations of the coarse-grained tracking approach. Furthermore, fine-grained systems do not impose the burden of focusing on avoiding the label creep problem on developers. By tracking information at fine granularity, such systems are seemingly more flexible and do not suffer from false alarms and label creep issues [Austin and Flanagan 2009] as coarse-grained systems do. Indeed, fine-grained systems such as JSFlow [Hedin et al. 2014] can often be used to secure existing, legacy applications; they only require developers to properly annotate the application.

This paper removes the division between fine- and coarse-grained dynamic IFC systems and the belief that they are fundamentally different. In particular, we show that *dynamic* fine-grained and coarse-grained IFC are equally expressive. Our work is inspired by the recent work of Rajani et al. [2017]; Rajani and Garg [2018], who prove similar results for *static* fine-grained and coarse-grained IFC systems. Specifically, they establish a semantics- and type-preserving translation from a coarse-grained IFC type system to a fine-grained one and vice-versa. We complete the picture by showing a similar result for dynamic IFC systems that additionally allow *introspection on labels* at run-time. While label introspection is meaningless in a static IFC system, in a dynamic IFC system this feature is key to both writing practical applications and mitigating the label creep problem [Stefan et al. 2017].

Using Agda, we formalize a traditional fine-grained system (in the style of [Austin and Flanagan 2009]) extended with label introspection primitives, as well as a coarse-grained system (in the style of [Stefan et al. 2017]). We then define and formalize modular semantics-preserving translations between them. Our translations are macro-expressible in the sense of Felleisen [1991].

We show that a translation from fine- to coarse-grained is possible when the coarse-grained system is equipped with a primitive that limits the scope of tainting (e.g., when reading sensitive data). In practice, this is not an imposing requirement since most coarse-grained systems rely on such primitives for compartmentalization. For example, Stefan et al. [2012, 2017], provide `toLabeled` blocks and threads for precisely this purpose. Dually, we show that the translation from coarse- to fine-grained is possible when the fine-grained system has a primitive **taint**$(\cdot)$ that relaxes precision to keep the *program counter label* synchronized when translating a program to the coarse-grained language. While this primitive is largely necessary for us to establish the coarse- to fine-grained translation, extending existing fine-grained systems with it is both secure and trivial.

Types:	$\tau ::=$	$\mathbf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mathscr{L} \mid \mathbf{Ref}\ \tau$
Labels:	$\ell, pc \in$	$\mathscr{L}$
Addresses:	$n \in$	$\mathbb{N}$
Environment:	$\theta \in$	$Var \rightharpoonup Value$
Raw Value:	$r ::=$	$() \mid (x.e, \theta) \mid \mathbf{inl}(v) \mid \mathbf{inr}(v) \mid (v_1, v_2) \mid \ell \mid n_\ell$
Value	$v ::=$	r^ℓ
Expressions:	$e ::=$	$x \mid \lambda x.e \mid e_1\ e_2 \mid () \mid \ell \mid \mathbf{inl}(e) \mid \mathbf{inr}(e) \mid \mathbf{case}(e, x.e_1, x.e_2)$
		$\mid (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \mathbf{getLabel} \mid \mathbf{labelOf}(e) \mid \mathbf{taint}(e_1, e_2)$
		$\mid \mathbf{new}(e) \mid !\ e \mid e_1 := e_2 \mid \mathbf{labelOfRef}(e) \mid e_1 \sqsubseteq^? e_2$
Configuration:	$c ::=$	$\langle \Sigma, e \rangle$
Store:	$\Sigma \in$	$(\ell : Label) \to Memory\ \ell$
Memory ℓ:	$M ::=$	$[\,] \mid r : M$

Fig. 1. Syntax of λ^{dFG}.

The implications of our results are multi-fold. The fine- to coarse-grained translation formally confirms an old OS-community hypothesis that it is possible to restructure a system into smaller compartments to address the label creep problem—indeed our translation is a (naive) algorithm for doing so. This translation also allows running legacy fine-grained IFC compatible applications atop coarse-grained systems like LIO. Dually, the coarse- to fine-grained translation allows developers building new applications in a fine-grained system to avoid the annotation burden of the fine-grained system by writing some of the code in the coarse-grained system and compiling it automatically to the fine-grained system with our translation.

The technical contributions of this paper are:

- A pair of semantics-preserving translations between traditional dynamic fine-grained and coarse-grained IFC systems equipped with label introspection (Theorems 3 and 5).
- Two different proofs of *termination-insensitive* non-interference (TINI) for each calculus: one is derived directly in the usual way (Theorems 1 and 2), while the other is recovered via our verified translation (Theorems 4 and 6).
- Mechanized Agda proofs of our results (~4,000 LOC)[1].

The rest of this paper is organized as follows. Our dynamic fine- and coarse-grained IFC calculi are introduced in Sections 2 and 3, respectively. We also prove their soundness guarantees (i.e., termination-insensitive non-interference). Section 4 presents the translation from the fine- to the coarse-grained calculus and recovers the non-interference of the former from the non-interference theorem of the latter. Section 5 has similar results in the other direction. Related work is described in Section 6 and Section 7 concludes the paper.

2 FINE-GRAINED CALCULUS

In order to compare in a rigorous way fine- and coarse-grained dynamic IFC techniques, we formally define the operational semantics of two λ-calculi that respectively perform fine- and coarse-grained IFC dynamically. Figure 1 shows the syntax of the dynamic fine-grained IFC calculus λ^{dFG}, which is inspired by Austin and Flanagan [Austin and Flanagan 2009] and extended with a standard (security unaware) type system $\Gamma \vdash e : \tau$ (omitted), sum and

[1] Artifact available at https://hub.docker.com/r/marcovassena/granularity/

product data types and security labels $\ell \in \mathscr{L}$ that form a lattice $(\mathscr{L}, \sqsubseteq)$.[2] In order to capture flows of information precisely at run-time, λ^{dFG} features *intrinsically labeled* values, written, r^{ℓ}, meaning that raw value r has security level ℓ. Compound values, e.g., pairs and sums, carry labels to tag the security level of each component, for example a pair containing a secret and a public boolean would be written $(\mathbf{true}^{H}, \mathbf{false}^{L})$.[3] Functional values are closures $(x.e, \theta)$, where x is the variable that binds the argument in the body of the function e and all other free variables are mapped to some labeled value in the environment θ. λ^{dFG} features a labeled partitioned stored, i.e., $\Sigma \in (\ell : \mathscr{L}) \to \mathit{Memory}\ \ell$, where $\mathit{Memory}\ \ell$ is the memory that contains values at security level ℓ. Each reference carries an additional label annotation that records the label of the memory it refers to—reference n_{ℓ} points to the n-th cell of the ℓ-labeled memory, i.e., $\Sigma(\ell)$. Notice that this label has nothing to do with the *intrinsic* label that decorates the reference itself. For example, a reference $(n_H)^{L}$ represents a secret reference in a public context, whereas $(n_L)^{H}$ represents a public reference in a secret context. Notice that there is no order invariant between those labels—in the latter case, the runtime monitor enforcing IFC prevents writing data to the reference to avoid *implicit flows*. A program can create, read and write a labeled reference via constructs $\mathbf{new}(e)$, $!e$ and $e_1 := e_2$ and inspect its subscripted label with the primitive $\mathbf{labelOfRef}(\cdot)$.

2.1 Dynamics

The operational semantics of λ^{dFG} includes a security monitor that propagates the label annotations of input values during program execution and assigns security labels to the result accordingly. The monitor prevents information leakage by stopping the execution of potentially leaky programs, which is reflected in the semantics by not providing reduction rules for the cases that may cause insecure information flow.[4] The relation $\langle \Sigma, e \rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v \rangle$ denotes the evaluation of program e with initial store Σ that terminates with labeled value v and final store Σ'. The environment θ stores the input values of the program and is extended with intermediate results during function application and case analysis. The subscript pc is the *program counter* label [Sabelfeld and Myers 2003]— it is a label that represents the security level of the context in which the expression is evaluated. The semantics employs the program counter label to 1) propagate and assign labels to values computed by a program and 2) prevent implicit flow leaks that exploit the control flow and the store (explained below).

In particular, when a program produces a value, the monitor tags the raw value with the program counter label in order to record the security level of the context in which it was computed. For this reason all the introduction rules for ground and compound types ([Unit,Label,Fun,Inl,Inr,Pair]) assign security level pc to the result. Other than that, these rules are fairly standard—we simply note that rule [Fun] creates a closure by capturing the current environment θ.

When the control flow of a program *depends* on some intermediate value, the program counter label is joined with the value's label so that the label of the final result will be tainted with the result of the intermediate value. For instance, consider case analysis, i.e., $\mathbf{case}\ e\ x.e_1\ x.e_2$. Rules [Case$_1$] and [Case$_2$] evaluate the scrutinee e to a value (either

[2] The lattice is arbitrary and fixed. In examples we will often use the two point lattice $\{L, H\}$, which only disallows secret to public flow of information, i.e., $H \not\sqsubseteq L$.

[3] We define the boolean type $\mathbf{bool} = \mathbf{unit} + \mathbf{unit}$, boolean values as raw values, i.e., $\mathbf{true} = \mathbf{inl}(()^{L})$, $\mathbf{false} = \mathbf{inr}(()^{L})$ and $\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 = \mathbf{case}\ e\ \_.e_1\ \_.e_2$.

[4] In this work we ignore leaks that exploit program termination. This is accounted for in the *termination insensitive* security condition satisfied by λ^{dFG} (Theorem 1).

$$\text{(Var)}\quad \langle \Sigma, x\rangle \Downarrow^{\theta}_{pc} \langle \Sigma, \theta(x) \sqcup pc\rangle \qquad \text{(Unit)}\quad \langle \Sigma, ()\rangle \Downarrow^{\theta}_{pc} \langle \Sigma, ()^{pc}\rangle \qquad \text{(Label)}\quad \langle \Sigma, \ell\rangle \Downarrow^{\theta}_{pc} \langle \Sigma, \ell^{pc}\rangle$$

$$\text{(Fun)}\quad \langle \Sigma, \lambda x.e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma, (x.e, \theta)^{pc}\rangle$$

$$\text{(App)}\quad \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', (x.e, \theta')^{\ell}\rangle \qquad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', v_2\rangle \qquad \langle \Sigma'', e\rangle \Downarrow^{\theta'[x\mapsto v_2]}_{pc \sqcup \ell} \langle \Sigma''', v\rangle}{\langle \Sigma, e_1\ e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma''', v\rangle}$$

$$\text{(Inl)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v\rangle}{\langle \Sigma, \mathbf{inl}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mathbf{inl}(v)^{pc}\rangle} \qquad \text{(Inr)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v\rangle}{\langle \Sigma, \mathbf{inr}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mathbf{inr}(v)^{pc}\rangle}$$

$$\text{(Case}_1\text{)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mathbf{inl}(v_1)^{\ell}\rangle \qquad \langle \Sigma', e_1\rangle \Downarrow^{\theta[x\mapsto v_1]}_{pc \sqcup \ell} \langle \Sigma'', v\rangle}{\langle \Sigma, \mathbf{case}(e, x.e_1, x.e_2)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', v\rangle}$$

$$\text{(Case}_2\text{)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mathbf{inr}(v_2)^{\ell}\rangle \qquad \langle \Sigma', e_2\rangle \Downarrow^{\theta[x\mapsto v_2]}_{pc \sqcup \ell} \langle \Sigma'', v\rangle}{\langle \Sigma, \mathbf{case}(e, x.e_1, x.e_2)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', v\rangle}$$

$$\text{(Pair)}\quad \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v_1\rangle \qquad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', v_2\rangle}{\langle \Sigma, (e_1, e_2)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', (v_1, v_2)^{pc}\rangle} \qquad \text{(Fst)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', (v_1, v_2)^{\ell}\rangle}{\langle \Sigma, \mathbf{fst}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v_1 \sqcup \ell\rangle}$$

$$\text{(Snd)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', (v_1, v_2)^{\ell}\rangle}{\langle \Sigma, \mathbf{snd}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v_2 \sqcup \ell\rangle} \qquad \text{(Taint)}\quad \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \ell^{\ell'}\rangle \qquad \ell' \sqsubseteq \ell \qquad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{\ell} \langle \Sigma'', v\rangle}{\langle \Sigma, \mathbf{taint}(e_1, e_2)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', v\rangle}$$

Fig. 2. Big-step semantics for λ^{dFG} (part I).

$\mathbf{inl}(v)^{\ell}$ or $\mathbf{inr}(v)^{\ell}$), add the value to the environment, i.e., $\theta[x \mapsto v]$, and then execute the appropriate branch with a program counter label tainted with v's security label, i.e., $pc \sqcup \ell$. As a result, the monitor tracks data dependencies across control flow constructs through the label of the result. Function application follows the same principle. In rule [App], since the first premise evaluates the function to some closure $(x.e, \theta')$ at security level ℓ, the third premise evaluates the body with program counter label raised to $pc \sqcup \ell$. The evaluation strategy is call-by-value: it evaluates the argument before the body in the second premise and binds the corresponding variable to its value in the environment of the closure, i.e., $\theta'[x \mapsto v_2]$. Notice that the security level of the argument is irrelevant at this

$$\text{(LabelOf)}\quad \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', r^{\ell}\rangle}{\langle \Sigma, \mathbf{labelOf}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \ell^{\ell}\rangle} \qquad \text{(GetLabel)}\quad \langle \Sigma, \mathbf{getLabel}\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', pc^{pc}\rangle$$

$$(\sqsubseteq^{?}\text{-T})\quad \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', {\ell_1}^{\ell_1'}\rangle \qquad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', {\ell_2}^{\ell_2'}\rangle \qquad \ell_1 \sqsubseteq \ell_2}{\langle \Sigma, e_1 \sqsubseteq^{?} e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', \mathbf{inl}(()^{pc})^{\ell_1' \sqcup \ell_2'}\rangle}$$

$$(\sqsubseteq^{?}\text{-F})\quad \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', {\ell_1}^{\ell_1'}\rangle \qquad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', {\ell_2}^{\ell_2'}\rangle \qquad \ell_1 \not\sqsubseteq \ell_2}{\langle \Sigma, e_1 \sqsubseteq^{?} e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', \mathbf{inr}(()^{pc})^{\ell_1' \sqcup \ell_2'}\rangle}$$

Fig. 3. Big-step semantics for λ^{dFG} (part II).

stage and that this is beneficial to not over-tainting the result: if the function never uses its argument then the label of the result depends exclusively on the program counter label, e.g., $(\lambda x.())\ y\ \Downarrow^{y \mapsto 42^{H}}_{L}\ ()^{L}$. The elimination rules for variables and pairs taint the label of the corresponding value with the program counter label for security reasons. In rules [Var,Fst,Snd] the notation, $v \sqcup \ell'$ upgrades the label of v with ℓ'—it is a shorthand for $r^{\ell \sqcup \ell'}$ with $v = r^{\ell}$. Intuitively, public values must be considered secret when the program counter is secret, for example $x\ \Downarrow^{x \mapsto ()^{L}}_{H}\ ()^{H}$.

Label Introspection. The λ^{dFG} calculus features primitives for label introspection, namely **getLabel**, **labelOf**$(\cdot)$ and $\sqsubseteq^{?}$—see Figure 3. These operations allow to respectively retrieve the current program counter label, obtain the label annotations of values, and compare two labels (inspecting labels at run-time is useful for controlling and mitigating the label creep problem).

Enabling label introspection raises the question of what label should be assigned to the label itself (in λ^{dFG} every value, including all label values, must be annotated with a label). As a matter of fact, labels can be used to encode secret information and thus careless label introspection may open the doors to information leakage [Stefan et al. 2017]. Notice that in λ^{dFG}, the label annotation on the result is computed by the semantics together with the result and thus it is as sensitive as the result itself (the label annotation on a value depends on the sensitivity of all values affecting the *control-flow* of the program up to the point where the result is computed). This motivates the design choice to protect each projected label with the label itself, i.e., ℓ^{ℓ} and pc^{pc} in rules [GetLabel] and [LabelOf] in Figure 2. We remark that this choice is consistent with previous work on coarse-grained IFC languages [Buiras et al. 2014; Stefan et al. 2017], but novel in the context of fine grained IFC.

Finally, primitive **taint**(e_1, e_2) temporarily raises the program counter label to the label given by the first argument in order to evaluate the second argument. The fine-to-coarse translation in Section 4 uses **taint**$(\cdot)$ to loosen the precision of λ^{dFG} in a controlled way and match the *coarse* approximation of our coarse-grained IFC calculus (λ^{dCG}) by upgrading the labels of intermediate values systematically. In rule [Taint], the constraint $\ell' \sqsubseteq \ell$ ensures

$$\textsc{New}\ \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', r^{\ell}\rangle \qquad n = |\Sigma'(\ell)|}{\langle \Sigma, \mathbf{new}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'[\ell \mapsto \Sigma'(\ell)[n \mapsto r]], (n_{\ell})^{pc}\rangle}$$

$$\textsc{Read}\ \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', {n_{\ell}}^{\ell'}\rangle \qquad \Sigma'(\ell)[n] = r}{\langle \Sigma, !e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', r^{\ell \sqcup \ell'}\rangle}$$

$$\textsc{Write}\ \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', {n_{\ell}}^{\ell_1}\rangle \qquad \ell_1 \sqsubseteq \ell \qquad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', r^{\ell_2}\rangle \qquad \ell_2 \sqsubseteq \ell}{\langle \Sigma, e_1 := e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma''[\ell \mapsto \Sigma''(\ell)[n \mapsto r]], {}^{pc}\rangle}$$

$$\textsc{LabelOfRef}\ \frac{\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', {n_{\ell}}^{\ell'}\rangle}{\langle \Sigma, \mathbf{labelOfRef}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \ell^{\ell \sqcup \ell'}\rangle}$$

Fig. 4. Big-step semantics for λ^{dFG} (references).

that the label of the nested context ℓ is at least as sensitive as the program counter label pc. In particular, this constraint ensures that the operational semantics have Property 1 (*"the label of the result is at least as sensitive as the program counter label"*) even with rule [Taint].

Property 1. *If* $\langle \Sigma, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', r^{\ell}\rangle$ *then* $pc \sqsubseteq \ell$.

Proof. By induction on the given evaluation derivation.

References. We now extend the semantics presented earlier with primitives that inspect, access and modify the labeled store via labeled references. See Figure 4. Rule [New] creates a reference n_{ℓ}, labeled with the security level of the initial content, i.e., ℓ in the ℓ-labeled memory $\Sigma(\ell)$ and updates the memory store accordingly.[5] Since the security level of the reference is as sensitive as the content, which is at least as sensitive as the program counter label by Property 1 ($pc \sqsubseteq \ell$) this operation does not leak information via *implicit flows*. When reading the content of reference n_{ℓ} at security level ℓ', rule [Read] retrieves the corresponding raw value from the n-th cell of the ℓ-labeled memory, i.e., $\Sigma'(\ell)[n] = r$ and upgrades its label to $\ell \sqcup \ell'$ since the decision to read from that particular reference depends on information at security level ℓ'. When writing to a reference the monitor performs security checks to avoid leaks via explicit or implicit flows. Rule [Write] achieves this by evaluating the reference, i.e., $(n_{\ell})^{\ell_1}$ and replacing its content with the value of the second argument, i.e., r^{ℓ_2}, under the conditions that the decision of "which" reference to update does not depend on data more sensitive than the reference itself, i.e., $\ell_1 \sqsubseteq \ell$ (not checking this would leak via an *implicit flow*)[6], and that the new content is no more sensitive than the reference itself, i.e., $\ell_2 \sqsubseteq \ell$ (not checking this would leak sensitive information to a less sensitive reference *explicitly*). Lastly, rule [LabelOfRef] retrieves the label of the reference and protects it with the label itself (as explained before) and taints it with the security

[5] $|M|$ denotes the length of memory M—memory indices start at 0.

[6] Notice that $pc \sqsubseteq \ell_1$ by Property 1, thus $pc \sqsubseteq \ell_1 \sqsubseteq \ell$ by transitivity. An *implicit flow* would occur if a reference is updated in a *high branch*, i.e., depending on the secret, e.g., **let** $x = \mathbf{new}(0)$ **in if** *secret* **then** $x :=$ 1 **else** ().

$$\text{(Value}_L\text{)}\ \frac{\ell \sqsubseteq L \qquad r_1 \approx_L r_2}{{r_1}^{\ell} \approx_L {r_2}^{\ell}} \qquad \text{(Value}_H\text{)}\ \frac{\ell_1 \not\sqsubseteq L \qquad \ell_2 \not\sqsubseteq L}{{r_1}^{\ell_1} \approx_L {r_2}^{\ell_2}} \qquad \text{(Unit)}\ () \approx_L () \qquad \text{(Label)}\ \ell \approx_L \ell$$

$$\text{(Closure)}\ \frac{e_1 \equiv_\alpha e_2 \qquad \theta_1 \approx_L \theta_2}{(e_1, \theta_1) \approx_L (e_2, \theta_2)} \qquad \text{(Inl)}\ \frac{v_1 \approx_L v_2}{\mathbf{inl}(v_1) \approx_L \mathbf{inl}(v_2)} \qquad \text{Inr}\ \frac{v_1 \approx_L v_2}{\mathbf{inr}(v_1) \approx_L \mathbf{inr}(v_2)}$$

$$\text{(Pair)}\ \frac{v_1 \approx_L v_1' \qquad v_2 \approx_L v_2'}{(v_1, v_2) \approx_L (v_1', v_2')} \qquad \text{(Ref}_L\text{)}\ \frac{\ell \sqsubseteq L}{n_\ell \approx_L n_\ell} \qquad \text{(Ref}_H\text{)}\ \frac{\ell_1 \not\sqsubseteq L \qquad \ell_2 \not\sqsubseteq L}{{n_1}_{\ell_1} \approx_L {n_2}_{\ell_2}}$$

Fig. 5. L-equivalence for λ^{dFG} values and raw values.

level of the reference, i.e., $\ell^{\ell \sqcup \ell'}$ to avoid leaks. Intuitively, the label of the reference, i.e., ℓ, depends also on data at security level ℓ' as seen in the premise.

Other Extensions. We consider λ^{dFG} equipped with references as sufficient foundation to study the relationship between fine-grained and coarse-grained IFC. We remark that extending it with other side-effects such as file operations, or other IO-operations would not change our claims in Section 4 and 5. The main reason for this is that, typically, handling such effects would be done at the same granularity in both IFC enforcements. For instance, when adding file operations, both fine- (e.g., [Broberg et al. 2013]) and coarse-grained (e.g., [Efstathopoulos et al. 2005; Krohn et al. 2007b; Russo et al. 2008; Stefan et al. 2011]) enforcements are likely to assign a single *flow-insensitive* label to each file in order to denote the sensitivity of its content. Then, those features could be handled *flow-insensitively* in both systems (e.g., [Myers et al. 2006; Pottier and Simonet 2002; Stefan et al. 2011; Vassena and Russo 2016]), in a manner similar to what we have just shown for references in λ^{dFG}.

2.2 Security

We now prove that λ^{dFG} is secure, i.e., it satisfies *termination insensitive non-interference* (TINI) [Goguen and Meseguer 1982; Volpano and Smith 1997]. Intuitively, the security condition says that no terminating λ^{dFG} program leaks information, i.e., changing secret inputs does not produce any publicly visible effect. The proof technique is standard and based on the notion of L-equivalence, written $v_1 \approx_L v_2$, which relates values (and similarly raw values, environments, stores and configurations) that are indistinguishable for an attacker at security level L. For clarity we use the 2-points lattice, assume that secret data is labeled with H and that the attacker can only observe data at security level L. Our mechanized proofs are parametric in the lattice and in the security level of the attacker. L-equivalence for values and raw-values is defined formally by mutual induction in Figure 5. Rule [Value$_L$] relates observable values, i.e., raw values labeled below the security level of the attacker. These values have the *same* observable label ($\ell \sqsubseteq L$) and related raw values, i.e., $r_1 \approx_L r_2$. Rule [Value$_H$] relates non-observable values, which may have different labels not below the attacker level, i.e., $\ell_1 \not\sqsubseteq L$ and $\ell_2 \not\sqsubseteq L$. In this case, the raw values can be arbitrary. Raw values are L-equivalent when they consist of the same ground value ([Unit,Label]), or are homomorphically related for compound values. For example, for the sum type the relation requires that both values are either a left or a right injection ([INL,INR]). In particular, closures are related if they contain the *same* function (up to α-renaming)[7] and L-equivalent environments, i.e., the environments are L-equivalent pointwise. Formally, $\theta_1 \approx_L \theta_2$ iff $Dom(\theta_1) \equiv Dom(\theta_2)$ and $\forall x.\theta_1(x) \approx_L \theta_2(x)$.

We define L-equivalence for stores pointwise, i.e., $\Sigma_1 \approx_L \Sigma_2$ iff for all labels $\ell \in \mathscr{L}$, $\Sigma_1(\ell) \approx_L \Sigma_2(\ell)$. Memory L-equivalence relates arbitrary ℓ-labeled memories if $\ell \not\sqsubseteq L$, and pointwise otherwise, i.e., $M_1 \approx_L M_2$ iff M_1 and M_2 are memories labeled with $\ell \sqsubseteq L$, $|M_1| = |M_2|$ and for all $n \in \{0\,..\,|M_1|-1\}$, $M_1[n] \approx_L M_2[n]$. Similarly, L-equivalence relates any two secret references (rule [REF_H]) but requires the same label and address for public references (rule [REF_L]).

We naturally lift L-equivalence to initial configurations, i.e., $c_1 \approx_L c_2$ iff $c_1 = \langle \Sigma_1, e_1 \rangle$, $c_2 = \langle \Sigma_2, e_2 \rangle$, $\Sigma_1 \approx_L \Sigma_2$ and $e_1 \equiv_\alpha e_2$, and final configurations, i.e., $c_1' \approx_L c_2'$ iff $c_1' = \langle \Sigma_1', v_1 \rangle$, $c_2' = \langle \Sigma_2', v_2 \rangle$ and $\Sigma_1' \approx_L \Sigma_2'$ and $v_1 \approx_L v_2$.

We now formally state and prove that λ^{dFG} semantics preserve L-equivalence under L-equivalent environments, i.e., *termination-insensitive non-interference* (TINI).

THEOREM 1 (λ^{dFG}-TINI). *If $c_1 \Downarrow_{pc}^{\theta_1} c_1'$, $c_2 \Downarrow_{pc}^{\theta_2} c_2'$, $\theta_1 \approx_L \theta_2$ and $c_1 \approx_L c_2$ then $c_1' \approx_L c_2'$.*

Proof. By induction on the derivations.

Dynamic language-based fine-grained IFC, of which λ^{dFG} is just a particular instance, represents an intuitive approach to tracking information flows in programs. Programmers annotate input values with labels that represent their sensitivity and a label-aware instrumented security monitor propagates those labels during execution and computes the result of the program together with a conservative approximation of its sensitivity. The next section describes an IFC monitor that tracks information flows at *coarse* granularity.

3 COARSE-GRAINED CALCULUS

One of the drawbacks of dynamic fine-grained IFC is that the programming model requires all input values to be explicitly and fully annotated with their security labels. Imagine a program with many inputs and highly structured data: it quickly becomes cumbersome, if not impossible, for the programmer to specify all the labels. The label of some inputs may be sensitive (e.g., passwords, pin codes, etc.), but the sensitivity of the rest may probably be irrelevant for the computation, yet a programmer must come up with appropriate labels for them as well. The programmer is then torn between two opposing risks: over-approximating the actual sensitivity can negatively affect execution (the monitor might stop secure programs), under-approximating the sensitivity can endanger security. Even worse, specifying many labels manually is error-prone and assigning the wrong security label to a piece of sensitive data can be catastrophic for security and completely defeat the purpose of IFC. Dynamic coarse-grained IFC represents an attractive alternative that requires fewer annotations, in particular it allows the programmer to label only the inputs that need to be protected.

Figure 6 shows the syntax of λ^{dCG}, a standard simply-typed λ-calculus extended with security primitives for dynamic coarse-grained IFC, inspired by Stefan et al. [2011] and adapted to use call-by-value instead of call-by-name to match λ^{dFG}. The calculus λ^{dCG} features both

[7] Symbol $\equiv_\alpha$ denotes α-equivalence. In our mechanized proofs we use De Bruijn indexes and syntactic equivalence.

Type:	$\tau ::=$	$\textbf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mathscr{L} \mid \textbf{LIO}\ \tau \mid \textbf{Labeled}\ \tau \mid \textbf{Ref}\ \tau$
Labels:	$\ell, pc \in$	$\mathscr{L}$
Addresses:	$n \in$	$\mathbb{N}$
Environment:	$\theta \in$	$Var \rightharpoonup Value$
Value:	$v ::=$	$() \mid (x.e, \theta) \mid \textbf{inl}(v) \mid \textbf{inr}(v) \mid (v_1, v_2) \mid \ell \mid \textbf{Labeled}\ \ell\ v \mid (t, \theta) \mid n_\ell$
Expression:	$e ::=$	$x \mid \lambda x.e \mid e_1\ e_2 \mid () \mid \ell \mid \textbf{inl}(e_1) \mid \textbf{inr}(e_2) \mid \textbf{case}(e, x.e_1, x.e_2)$
		$\mid (e_1, e_2) \mid \textbf{fst}(e) \mid \textbf{snd}(e) \mid e_1 \sqsubseteq^? e_2 \mid t$
Thunk	$t ::=$	$\textbf{return}(e) \mid \textbf{bind}(e, x.e) \mid \textbf{unlabel}(e) \mid \textbf{toLabeled}(e) \mid \textbf{labelOf}(e)$
		$\mid \textbf{getLabel} \mid \textbf{taint}(e) \mid \textbf{new}(e) \mid !\ e \mid e_1 := e_2 \mid \textbf{labelOfRef}(e)$
Type System:	$\Gamma \vdash e : \tau$	
Configuration:	$c ::=$	$\langle \Sigma, pc, e \rangle$
Store:	$\Sigma \in$	$(\ell : Label) \to Memory\ \ell$
Memory ℓ:	$M ::=$	$[\,] \mid v : M$

Fig. 6. Syntax of λ^{dCG}.

standard (unlabeled) values and *explicitly labeled* values. For example, **Labeled** H **true** represents a secret boolean value of type **Labeled bool**.[8] The type constructor **LIO** encapsulates a security state monad, whose state consists of a labeled store and the program counter label. In addition to standard $\textbf{return}(\cdot)$ and $\textbf{bind}(\cdot)$ constructs, the monad provides primitives that regulate the creation and the inspection of labeled values, i.e., $\textbf{toLabeled}(\cdot)$, $\textbf{unlabel}(\cdot)$ and $\textbf{labelOf}(\cdot)$, and the interaction with the labeled store, allowing the creation, reading and writing of labeled references n_ℓ through the constructs $\textbf{new}(e)$, $!e$, $e_1 := e_2$, respectively. It also features an operator to query if a label flows to another, written $\ell_1 \sqsubseteq^? \ell_2$. The primitives of the **LIO** monad are listed in a separate sub-category of expressions called *thunk*. Intuitively, a thunk is just a description of a stateful computation, which only the top-level security monitor can execute—a *thunk closure*, i.e., (t, θ), provides a way to suspend computations.

3.1 Dynamics

In order to track information flows dynamically at coarse granularity, λ^{dCG} employs a technique called *floating-label*, which was originally developed for IFC operating systems (e.g., [Zeldovich et al. 2006, 2008]) and that was later applied in a language-based setting. In this technique, throughout a program's execution, the program counter *floats* above the label of any value observed during program execution and thus represents (an upper-bound on) the sensitivity of all the values that are not explicitly labeled. For this reason, λ^{dCG} stores the program counter label in the program configuration, so that the primitives of the **LIO** monad can control it explicitly (in technical terms the program counter is *flow-sensitive*, i.e., it may assume different values in the final configuration depending on the control flow of the program).[9]

[8] As in λ^{dFG}, we define **bool** = **unit** + **unit** and **if** e **then** e_1 **else** e_2 = **case** e $\_.e_1$ $\_.e_2$. Unlike λ^{dFG} values, λ^{dCG} values are not intrinsically labeled, thus we encode boolean constants simply as **true** = $\textbf{inl}()$ and **false** = $\textbf{inr}()$.

[9] In contrast, we consider λ^{dFG}'s program counter *flow-insensitive* because it is part of the evaluation judgment and its value changes only inside nested judgments.

Like λ^{dFG}, the operational semantics of λ^{dCG} consists of a security monitor that fully evaluates secure programs but prevents the execution of insecure programs and similarly enforces *termination-insensitive* non-interference (Theorem 2). Figure 7 shows the big-step operational semantics of λ^{dCG} in two parts: (i) a top-level security monitor for monadic programs and (ii) a straightforward call-by-value side-effect-free semantics for pure expressions. The semantics of the security monitor is further split into two mutually recursive reduction relations, one for arbitrary expressions (Fig. 7a) and one specific to thunks (Fig. 7c). These constitute the *forcing* semantics of the monad, which reduce a thunk to a pure value and perform side-effects. In particular, given the initial store Σ, program counter label pc, expression e of type **LIO** τ for some type τ and input values θ (which may or may not be labeled), the monitor executes the program, i.e., $\langle \Sigma, pc, e \rangle \Downarrow^{\theta} \langle \Sigma', pc', v \rangle$ and gives an updated store Σ', updated program counter pc' and a final value v of type τ, which also might not be labeled. The execution starts with rule [Force], which reduces the pure expression to a thunk closure, i.e., (t, θ') and then forces the thunk t in its environment θ' with the thunk semantics. The pure semantics is fairly standard—we report some selected rules in Fig. 7b for comparison with λ^{dFG}. A pure reduction, written $e \Downarrow^{\theta} v$, evaluates an expression e with an appropriate environment θ to a pure value v. Notice that, unlike λ^{dFG}, all reduction rules of the pure semantics ignore security, even those that affect the control flow of the program, e.g., rule [App]: they do not feature the program counter label or label annotations. They are also *pure*—they do not have access to the store, thus only the security monitor needs to protect against *implicit flows*.

If the pure evaluation reaches a side-effectful computation, i.e., thunk t, it *suspends* the computation by creating a thunk closure that captures the current environment θ (see rule [Thunk]). Notice that *thunk closures* and *function closures* are distinct values created by different rules, [Thunk] and [Fun] respectively.[10] Function application succeeds only when the function evaluates to a function closure (rule [App]). In the thunk semantics, rule [Return] evaluates a pure value embedded in the monad via **return**$(\cdot)$ and leaves the state unchanged, while rule [Bind] executes the first computation with the forcing semantics, binds the result in the environment i.e., $\theta[x \mapsto v_1]$, passes it on to the second computation together with the updated state and returns the final result and state. Rule [Unlabel] is interesting. Following the *floating-label* principle, it returns the value wrapped inside the labeled value, i.e., v, and raises the program counter with its label, i.e., $pc \sqcup \ell$, to reflect the fact that new data at security level ℓ is now in scope.

Floating-label based coarse-grained IFC systems like **LIO** suffer from the *label creep* problem, which occurs when the program counter gets over-tainted, e.g., because too many secrets have unlabeled, to the point that no useful further computation can be performed. Primitive **toLabeled**$(\cdot)$ provides a mechanism to address this problem by (i) creating a separate context where some sensitive computation can take place and (ii) restoring the original program counter label afterwards. Rule [ToLabeled] formalizes this idea. Notice that the result of the nested sensitive computation, i.e., v, cannot be simply returned to the lower context—that would be a leak, so **toLabeled**$(\cdot)$ wraps that piece of information in a labeled value protected by the final program counter of the sensitive computation, i.e., **Labeled** pc' v.[11] Furthermore, notice that pc', the label that tags the result v, is as sensitive as the result itself because the final program counter depends on all the **unlabel**$(\cdot)$

[10] It would have also been possible to define thunk values in terms of function closures using explicit suspension and an opaque wrapper, e.g., **LIO** $(\_.t, \theta)$.

[11] Stefan et al. [2017] have proposed an alternative flow-insensitive primitive, i.e., **toLabeled**(ℓ, e), which labels the result with the user-assigned label ℓ. The semantics of λ^{dFG} forced us to use **toLabeled**(e).

operations performed to compute the result. This motivates why primitive **labelOf**$(\cdot)$ does not simply project the label from a labeled value, but additionally taints the program counter with the label itself in rule [LabelOf]–a label in a labeled value has sensitivity equal to the label itself, thus the program counter label rises to accommodate reading new sensitive data.

Lastly, rule [GetLabel] returns the value of the program counter, which does not rise (because $pc \sqcup pc = pc$), and rule [Taint] simply taints the program counter with the given label and returns unit (this primitive matches the functionality of **taint**$(\cdot)$ in λ^{dFG}). Note that, in λ^{dCG}, **taint**$(\cdot)$ takes *only* the label with which the program counter must be tainted whereas, in λ^{dFG}, it additionally requires the expression that must be evaluated in the tainted environment. This difference highlights the *flow-sensitive* nature of the program counter label in λ^{dCG}.

References. λ^{dCG} features *flow-insensitive* labeled references similar to λ^{dFG} and allows programs to create, read, update and inspect the label inside the **LIO** monad (see Figure 8). The API of these primitives takes explicitly labeled values as arguments, by making explicit at the type level, the tagging that occurs in memory, which was left implicit in previous work [Stefan et al. 2017]. Rule [New] creates a reference labeled with the same label annotation as that of the labeled value it receives as an argument, and checks that $pc \sqsubseteq \ell$ in order to avoid implicit flows. Rule [Read] retrieves the content of the reference from the ℓ-labeled memory and returns it. Since this brings data at security level ℓ in scope, the program counter is tainted accordingly, i.e., $pc \sqcup \ell$. Rule [Write] performs security checks analogous to those in λ^{dFG} and updates the content of a given reference and rule [LabelOfRef] returns the label on a reference and taints the context accordingly.

We conclude this section by noting that the forcing and the thunk semantics of λ^{dCG} satisfy Property 2 (*"the final value of the program counter is at least as sensitive as the initial value"*).

Property 2.

- *If* $\langle \Sigma, pc, e\rangle \Downarrow^{\theta} \langle \Sigma', pc', v\rangle$ *then* $pc \sqsubseteq pc'$.
- *If* $\langle \Sigma, pc, t\rangle \Downarrow^{\theta} \langle \Sigma', pc', v\rangle$ *then* $pc \sqsubseteq pc'$.

Proof. By mutual induction on the given evaluation derivations.

3.2 Security

We now prove that λ^{dCG} is secure, i.e., it satisfies *termination-insensitive non-interference.* The meaning of the security condition is intuitively similar to that presented in Section 2.2 for λ^{dFG}— when secret inputs are changed, terminating programs do not produce any publicly observable effect—and based on a similar indistinguishability relation. Figure 9 presents the definition of L-equivalence for the interesting cases only. Firstly, L-equivalence for λ^{dCG} labeled values relates public and secret values analogously to λ^{dFG} values. Specifically, rule [Labeled$_L$] relates public labeled values that share the *same* observable label ($\ell \sqsubseteq L$) and contain related values, i.e., $v_1 \approx_L v_2$, while rule [Labeled$_H$] relates secret labeled values, with arbitrary sensitivity labels not below L ($\ell_1 \not\sqsubseteq L$ and $\ell_2 \not\sqsubseteq L$) and contents. Secondly, L-equivalence relates standard (unlabeled) values homomorphically. For example, values of the sum type are related only as follows: $\mathbf{inl}(v_1) \approx_L \mathbf{inl}(v_1')$ iff $v_1 \approx_L v_1'$ and $\mathbf{inr}(v_2) \approx_L \mathbf{inr}(v_2')$ iff $v_2 \approx_L v_2'$. Closures and thunks are related if the function and the monadic computations are α-equivalent and their environments are related, i.e., $\theta_1 \approx_L \theta_2$ iff $Dom(\theta_1) \equiv Dom(\theta_2)$ and $\forall x.\theta_1(x) \approx_L \theta_2(x)$. L-equivalence relates labeled references, memories and stores analogously to λ^{dFG}. Related initial configurations have related stores, equal program counters, and

α-equivalent programs, i.e., $c_1 \approx_L c_2$ iff $c_1 = \langle \Sigma_1, pc_1, e_1 \rangle$, $c_2 = \langle \Sigma_2, pc_2, e_2 \rangle$, $\Sigma_1 \approx_L \Sigma_2$, $pc_1 \equiv pc_2$, and $e_1 \equiv_\alpha e_2$. L-equivalence relates final configurations in which either 1) [Pc_L] the attacker can observe the same program counter $pc \sqsubseteq L$ in *both* configurations, which then carry related stores and values, or 2) [Pc_H] the value of the program counter in *both* configuration is not below the attacker level, which thus contain arbitrary values and related stores.

We now formally state and prove that λ^{dCG} semantics preserve L-equivalence under L-equivalent environments, i.e., *termination-insensitive non-interference* (TINI).

Theorem 2 (λ^{dCG}-TINI). *If* $c_1 \Downarrow^{\theta_1} c_1'$, $c_2 \Downarrow^{\theta_2} c_2'$, $\theta_1 \approx_L \theta_2$ *and* $c_1 \approx_L c_2$ *then* $c_1' \approx_L c_2'$.

Proof. By induction on the derivations.

At this point, we have formalized two calculi—λ^{dFG} and λ^{dCG}—that perform dynamic IFC at *fine* and *coarse* granularity, respectively. While they have some similarities, i.e., they are both functional languages that feature labeled annotated data, references and label introspection primitives, and ensure a termination-insensitive security condition, they also have striking differences. First and foremost, they differ in the number of label annotations—pervasive in λ^{dFG} and optional in λ^{dCG}—with significant implications for the programming model and usability. Second, they differ in the nature of the program counter, *flow-insensitive* in λ^{dFG} and *flow-sensitive* in λ^{dCG}. Third, they differ in the way they deal with side-effects—λ^{dCG} allows side-effectful computations exclusively inside the monad, while λ^{dFG} is *impure*, i.e., any λ^{dFG} expression can modify the state. This difference affects the effort required to implement a system that performs language-based fine- and coarse-grained dynamic IFC. In fact, several coarse-grained IFC languages [Buiras et al. 2015; Jaskelioff and Russo 2011; Russo 2015; Russo et al. 2008; Schmitz et al. 2018; Tsai et al. 2007] have been implemented as an embedded domain specific language (EDSL) in a Haskell library with little effort, exploiting the strict control that the host language provides on side-effects. Adapting an existing language to perform fine-grained IFC requires major engineering effort, because several components (all the way from the parser to the runtime system) must be adapted to be label-aware.

In the next two sections we show that—despite their differences—these two calculi are, in fact, equally expressive.

4 FINE- TO COARSE-GRAINED PROGRAM TRANSLATION

This section presents a provably semantics-preserving program translation from the fine-grained dynamic IFC calculus λ^{dFG} to the coarse-grained calculus λ^{dCG}. At a high level, the translation performs two tasks (i) it embeds the *intrinsic* label annotation of λ^{dFG} values into an *explicitly* labeled λ^{dCG} value via the **Labeled** type constructor and (ii) it restructures λ^{dFG} *side-effectful* expressions into *monadic operations* inside the **LIO** monad. Our type-driven approach starts by formalizing this intuition in the function $\langle\!\langle \cdot \rangle\!\rangle$, which maps the λ^{dFG} type τ to the corresponding λ^{dCG} type $\langle\!\langle \tau \rangle\!\rangle$ (see Figure 10). The function is defined by induction on types and recursively adds the **Labeled** type constructor to each existing λ^{dFG} type constructor. For the function type $\tau_1 \rightarrow \tau_2$, the result is additionally monadic, i.e., $\langle\!\langle \tau_1 \rangle\!\rangle \rightarrow \mathbf{LIO} \langle\!\langle \tau_2 \rangle\!\rangle$. This is because the function's body in λ^{dFG} may have side-effects. The translation for values (Figure 11) is straightforward. Each λ^{dFG} label tag becomes the label annotation in a λ^{dCG} labeled value. The translation is homomorphic in constructors on raw values. The translation converts a λ^{dFG} function closure into a λ^{dCG} closure by translating the body of the function to a thunk, i.e., $\langle\!\langle e \rangle\!\rangle$, (see below) and translating the environment pointwise, i.e., $\langle\!\langle \theta \rangle\!\rangle = \lambda x.\langle\!\langle \theta(x) \rangle\!\rangle$.

Expressions. We show the translation of λ^{dFG} expressions to λ^{dCG} monadic thunks in Figure 12. We use the standard **do** notation for readability.[12] First, notice that the translation of all constructs occurs inside a **toLabeled**$(\cdot)$ block. This achieves two goals, (i) it ensures that the value that results from a translated expression is *explicitly* labeled and (ii) it creates an isolated nested context where the translated thunk can execute without raising the program counter label at the top level. Inside the **toLabeled**$(\cdot)$ block, the program counter label may rise, e.g., when some intermediate result is unlabeled, and the translation relies on **LIO**'s floating-label mechanism to track dependencies between data of different security levels. In particular, we will show later that the value of the program counter label at the end of each nested block coincides with the label annotation of the λ^{dFG} value that the original expression evaluates to. For example, introduction forms of ground values (unit, labels, and functions) are simply returned inside the **toLabeled**$(\cdot)$ block so that they get tagged with the current value of the program counter label just as in the corresponding λ^{dFG} introduction rules ([Label,Unit,Fun]). Introduction forms of compounds values such as **inl**(e), **inr**(e) and (e_1, e_2) follow the same principle. The translation simply nests the translations of the nested expressions inside the same constructor, without raising the program counter label. This matches the behavior of the corresponding λ^{dFG} rules [Inl,Inr,Pair].[13] For example, the λ^{dFG} reduction $((), ()) \Downarrow^{\varnothing}_{L} (()^{L}, ()^{L})^{L}$ maps to a λ^{dCG} reduction that yields **Labeled** L (**Labeled** L $()$, **Labeled** L $()$) when started with program counter label L.

The translation of variables gives some insight into how the λ^{dCG} floating-label mechanism can simulate λ^{dFG}'s tainting approach. First, the type-driven approach set out in Figure 10 demands that functions take only labeled values as arguments, so the variables in the source program are always associated to a labeled value in the translated program. The values that correspond to these variables are stored in the environment θ and translated separately, e.g., if $\theta(x) = r^{\ell}$ in λ^{dFG}, then x gets bound to $\langle\!\langle r^{\ell} \rangle\!\rangle =$ **Labeled** $\ell \langle\!\langle r \rangle\!\rangle$ when translated to λ^{dCG}. Thus, the translation converts a variable, say x, to **toLabeled**(**unlabel**(x)), so that its label gets tainted with the current program counter label. More precisely, **unlabel**(x) retrieves the labeled value associated with the variable, i.e., **Labeled** $\ell \langle\!\langle r \rangle\!\rangle$, taints the program counter with its label to make it $pc \sqcup \ell$, and returns the content, i.e., $\langle\!\langle r \rangle\!\rangle$. Since **unlabel**$(x)$ occurs inside a **toLabeled**$(\cdot)$ block, the code above results in **Labeled** $(pc \sqcup \ell)\langle\!\langle r \rangle\!\rangle$ when evaluated, matching precisely the tainting behavior of λ^{dFG} rule [Var], i.e., $x \Downarrow_{pc}^{\theta[x \mapsto r^{\ell}]} r^{pc \sqcup \ell}$.

The elimination forms for other types (function application, pair projections and case analysis) follow the same approach. For example, the code that translates a function application $e_1\ e_2$ first executes the code that computes the translated function, i.e., $lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle$, then the code that computes the argument, i.e., $lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle$ and then retrieves the function from the first labeled value, i.e., $v_1 \leftarrow$ **unlabel**(lv_1).[14] The function v_1 applied to the labeled argument lv_2 gives a computation that gets executed and returns a labeled value lv that gets unlabeled to expose the final result (the surrounding **toLabeled**$(\cdot)$ wraps it again

[12] Syntax **do** $x \leftarrow e_1; e_2$ desugars to **bind**$(e_1, x.e_2)$ and syntax $e_1; e_2$ to **bind**$(e_1, \_.e_2)$.
[13] We name a variable lv if it gets bound to a labeled value, i.e., to indicate that the variable has type **Labeled** τ.
[14] Notice that it is incorrect to unlabel the function before translating the argument, because that would taint the program counter label, which would raise at level, say $pc \sqcup \ell$, and affect the code that translates the argument, which was to be evaluated with program counter label equal to pc by the original *flow-insensitive* λ^{dFG} rule [App] for function application.

right away). The translation of case analysis is analogous. The translation of pair projections first converts the λ^{dFG} pair into a computation that gives a λ^{dCG} labeled pair of labeled values, say **Labeled** ℓ (**Labeled** $\ell_1\langle\!\langle r_1\rangle\!\rangle$, **Labeled** $\ell_2\langle\!\langle r_2\rangle\!\rangle$) and removes the label tag on the pair via **unlabel**, thus raising the program counter label to $pc \sqcup \ell$. Then, it projects the appropriate component and unlabels it, thus tainting the program counter label even further with the label of either the first or the second component. This coincides with the tainting mechanism of λ^{dFG} for projection rules, e.g., in rule [Fst] where $\mathbf{fst}(e) \Downarrow^{\theta}_{pc} \ {r_1}^{pc \sqcup \ell \sqcup \ell_1}$ if $e \Downarrow^{\theta}_{pc} \ ({r_1}^{\ell_1}, {r_2}^{\ell_2})^{\ell}$.

Lastly, translating $\mathbf{taint}(e_1, e_2)$ requires (i) translating the expression e_1 that gives the label, (ii) using $\mathbf{taint}(\cdot)$ from λ^{dCG} to explicitly taint the program counter label with the label that e_1 gives, and (iii) translating the second argument e_2 to execute in the tainted context and unlabeling the result. The construct $\mathbf{labelOf}(e)$ of λ^{dFG} uses the corresponding λ^{dCG} primitive applied on the corresponding labeled value, say **Labeled** $\ell\langle\!\langle r\rangle\!\rangle$, obtained from the translated expression. Notice that $\mathbf{labelOf}(\cdot)$ taints the program counter label in λ^{dCG}, which rises to $pc \sqcup \ell$, so the code just described results in **Labeled** $(pc \sqcup \ell)\ \ell$, which corresponds to the translation of the result in λ^{dFG}, i.e., $\langle\!\langle \ell^{\ell}\rangle\!\rangle =$ **Labeled** $\ell\ \ell$ because $pc \sqcup \ell \equiv \ell$, since $pc \sqsubseteq \ell$ from Property 1. The translation of **getLabel** follows naturally by simply wrapping λ^{dCG}'s **getLabel** inside a $\mathbf{toLabeled}(\cdot)$, which correctly returns the program counter label labeled with itself, i.e., **Labeled** $pc\ pc$.

Note on Environments. λ^{dFG} and λ^{dCG} semantics feature an environment θ for input values that gets extended with intermediate values during program evaluation and that may be captured inside a closure. Unfortunately, this capturing behavior is undesirable for our program translation. The program translation defined above introduces temporary auxiliary variables that carry the value of intermediate results, e.g., the labeled value obtained from running a computation that translates some λ^{dFG} expression. When the translated program is executed, these values end up in the environment, e.g., by means of rules [App] and [Bind], and mix with the input values of the source program and output values as well, thus complicating the correctness statement of the translation, which now has to account for those extra variables as well. In order to avoid this nuisance, we employ a special form of weakening that allows shrinking the environment at run-time and removing spurious values that are not needed in the rest of the program. In particular, expression *wken* $\overline{x}$ e has the same type as e if variables $\overline{x}$ are not free in e. At run-time, *wken* $\overline{x}$ e, evaluates e in an environment from which variables $\overline{x}$ have been dropped, so they do not get captured in closures created during the execution of e. Formally:

$$\frac{\Gamma \setminus \overline{x} \vdash e : \tau}{\Gamma \vdash \mathit{wken}\ \overline{x}\ e : \tau} \qquad \text{(Wken)}\ \frac{e \Downarrow^{\theta \setminus \overline{x}} \ v}{\mathit{wken}\ \overline{x}\ e \Downarrow^{\theta} \ v}$$

Rule [Wken] is part of the pure semantics of λ^{dCG}—the semantics of λ^{dFG} includes an analogous rule (the issue of contaminated environments arises in the translations in both directions so both calculi feature *wken*). We remark that this expedient is not essential—we can avoid it by slightly complicating the correctness statement to explicitly account for those extra variables. Nor is this expedient particularly interesting. In fact, we omit *wken* from the code of the program translations to avoid clutter (our mechanization includes *wken* in the appropriate places).

References. Figure 13 shows the program translation of λ^{dFG} primitives that access the store via references. The translation of λ^{dFG} values wraps references in λ^{dCG} labeled values (Figure 11), so the translations of Figure 13 take care of boxing and unboxing references. The translation of $\mathbf{new}(e)$ has a top-level $\mathbf{toLabeled}(\cdot)$ block that simply translates the content $(lv \leftarrow \langle\!\langle e \rangle\!\rangle)$ and puts it in a new reference $(\mathbf{new}(lv))$. λ^{dCG} assigns the label of the translated content to the new reference using the λ^{dCG} rule [New] (Figure 8), which also gets labeled with the original program counter label[15], just as in the λ^{dFG} rule [New] (Figure 4). In λ^{dFG}, rule [Read] reads from a reference $n_\ell{}^{\ell'}$ at security level ℓ' that points to the ℓ-labeled memory, and returns the content $\Sigma(\ell)[n]^{\ell \sqcup \ell'}$ at level $\ell \sqcup \ell'$. Similarly, the translation creates a $\mathbf{toLabeled}(\cdot)$ block that executes to get a labeled reference $lr = \mathbf{Labeled}\ \ell'\ n_\ell$, extracts the reference n_ℓ $(r \leftarrow \mathbf{unlabel}(lr))$ tainting the program counter label with ℓ', and then reads the reference's content further tainting the program counter label with ℓ as well. The code that translates and updates a reference consists of two $\mathbf{toLabeled}(\cdot)$ blocks. The first block is responsible for the update: it extracts the labeled reference and the labeled new content (lr and lv resp.), extracts the reference from the labeled value $(r \leftarrow \mathbf{unlabel}(lr))$ and updates it $(r := lv)$. The second block, $\mathbf{toLabeled}(\mathbf{return}())$, returns unit at security level pc, i.e., $\mathbf{Labeled}\ pc\ ()$, similar to the λ^{dFG} rule [Write]. The translation of $\mathbf{labelOfRef}(e)$ extracts the reference and projects its label via the λ^{dCG} primitive $\mathbf{labelOfRef}(\cdot)$, which additionally taints the program counter with the label itself, similar to the λ^{dFG} rule [LabelOfRef].

4.1 Correctness

In this section, we establish some desirable properties of the λ^{dFG}-to-λ^{dCG} translation defined above. These properties include type and semantics preservation as well as recovery of non-interference—a meta criterion that rules out a class of semantically correct (semantics preserving), yet elusive translations that do not preserve the meaning of security labels [Barthe et al. 2007; Rajani and Garg 2018].

We start by showing that the program translation preserves typing. The type translation for typing contexts Γ is pointwise, i.e., $\langle\!\langle \Gamma \rangle\!\rangle = \lambda x.\langle\!\langle \Gamma(x) \rangle\!\rangle$.

Lemma 4.1 (Type Preservation). *Given a well-typed λ^{dFG} expression, i.e., $\Gamma \vdash e : \tau$, the translated λ^{dCG} expression is also well-typed, i.e., $\langle\!\langle \Gamma \rangle\!\rangle \vdash \langle\!\langle e \rangle\!\rangle : \mathbf{LIO} \langle\!\langle \tau \rangle\!\rangle$.*

Proof. By induction on the given typing derivation. □

The main correctness criterion for the translation is semantics preservation. Intuitively, proving this theorem ensures that the program translation preserves the meaning of secure λ^{dFG} programs when translated and executed with λ^{dCG} semantics (under a translated environment). In the theorem below, the translation of stores and memories is pointwise, i.e., $\langle\!\langle \Sigma \rangle\!\rangle = \lambda \ell.\langle\!\langle \Sigma(\ell) \rangle\!\rangle$, and $\langle\!\langle [] \rangle\!\rangle = []$ and $\langle\!\langle r : M \rangle\!\rangle = \langle\!\langle r \rangle\!\rangle : \langle\!\langle M \rangle\!\rangle$ for each ℓ-labeled memory M. Furthermore, notice that in the translation, the initial and final program counter labels are the same. This establishes that the program translation preserves the flow-insensitive program counter label of λ^{dFG} (by means of primitive $\mathbf{toLabeled}(\cdot)$).

Theorem 3 (Semantics Preservation of $\langle\!\langle \cdot \rangle\!\rangle : \lambda^{dFG} \to \lambda^{dCG}$). *Given a well-typed λ^{dFG} program $\langle \Sigma, e \rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v \rangle$, then $\langle \langle\!\langle \Sigma \rangle\!\rangle, pc, \langle\!\langle e \rangle\!\rangle \rangle \Downarrow^{\langle\!\langle \theta \rangle\!\rangle} \langle \langle\!\langle \Sigma' \rangle\!\rangle, pc, \langle\!\langle v \rangle\!\rangle \rangle$.*

Proof. By induction on the given evaluation derivation using basic properties of the security lattice and of the translation function.

[15] The nested block does not execute any $\mathbf{unlabel}(\cdot)$ nor $\mathbf{taint}(\cdot)$.

Recovery of non-interference. We conclude this section by constructing a proof of termination-insensitive non-interference for λ^{dFG} (Theorem 1) from the corresponding theorem for λ^{dCG} (Theorem 2), using the semantics preservation of the translation, together with a property that the translation preserves L-equivalence. Doing so ensures that the meaning of labels is preserved by the translation [Barthe et al. 2007; Rajani and Garg 2018]. In the absence of such an artifact, one could devise a semantics-preserving translation that simply does not use the security features of the target language. While technically correct (i.e., semantics preserving), the translation would not be meaningful from the perspective of security.[16]

The theorem requires a helping lemma that L-equivalence is preserved by the translation. In the following, we define the translation for *initial* configuration as $\langle\!\langle c \rangle\!\rangle^{pc} = \langle \langle\!\langle \Sigma \rangle\!\rangle, pc, \langle\!\langle e \rangle\!\rangle \rangle$ if $c = \langle \Sigma, e \rangle$, and for *final* configurations $\langle\!\langle c \rangle\!\rangle^{pc} = \langle \langle\!\langle \Sigma \rangle\!\rangle, pc, \langle\!\langle v \rangle\!\rangle \rangle$ if $c = \langle \Sigma, v \rangle$.

Lemma 4.2. *For all values, raw values, environments and configurations:*

- $v_1 \approx_L v_2$ *if and only if* $\langle\!\langle v_1 \rangle\!\rangle \approx_L \langle\!\langle v_2 \rangle\!\rangle$.
- $r_1 \approx_L r_2$ *if and only if* $\langle\!\langle r_1 \rangle\!\rangle \approx_L \langle\!\langle r_2 \rangle\!\rangle$
- $\theta_1 \approx_L \theta_2$ *if and only if* $\langle\!\langle \theta_1 \rangle\!\rangle \approx_L \langle\!\langle \theta_2 \rangle\!\rangle$
- *Let* c_1 *and* c_2 *be initial configurations, then for all* pc, $c_1 \approx_L c_2$ *if and only if* $\langle\!\langle c_1 \rangle\!\rangle^{pc} \approx_L \langle\!\langle c_2 \rangle\!\rangle^{pc}$.
- *Let* $c_1' = \langle \Sigma, {r_1}^{\ell_1} \rangle$, $c_2' = \langle \Sigma_2, {r_2}^{\ell_2} \rangle$, *if* $pc \sqsubseteq \ell_1$, $pc \sqsubseteq \ell_2$ *and* $\langle\!\langle c_1' \rangle\!\rangle^{pc} \approx_L \langle\!\langle c_2' \rangle\!\rangle^{pc}$ *then* $c_1' \approx_L c_2'$.

Proof. By mutual induction and using injectivity of the translation function in the *if* direction.

Theorem 4 (λ^{dFG}-TINI via $\langle\!\langle \cdot \rangle\!\rangle$). *If* $c_1 \Downarrow_{pc}^{\theta_1} c_1'$, $c_2 \Downarrow_{pc}^{\theta_2} c_2'$, $\theta_1 \approx_L \theta_2$ *and* $c_1 \approx_L c_2$ *then* $c_1' \approx_L c_2'$.

Proof. We start by applying the fine to coarse grained program translation to the initial configurations and input values. By Theorem 3 (semantics preservation), we derive the corresponding λ^{dCG} reductions, i.e., $\langle\!\langle c_1 \rangle\!\rangle^{pc} \Downarrow^{\langle\!\langle \theta \rangle\!\rangle} \langle\!\langle c_1' \rangle\!\rangle^{pc}$ and $\langle\!\langle c_2 \rangle\!\rangle^{pc} \Downarrow^{\langle\!\langle \theta \rangle\!\rangle} \langle\!\langle c_2' \rangle\!\rangle^{pc}$. Then, we lift L-equivalence for initial configurations and input values to their translation (Lemma 4.2), i.e., $\langle\!\langle c_1 \rangle\!\rangle^{pc} \approx_L \langle\!\langle c_2 \rangle\!\rangle^{pc}$ and $\langle\!\langle \theta_1 \rangle\!\rangle \approx_L \langle\!\langle \theta_2 \rangle\!\rangle$ and obtain $\langle\!\langle c_1' \rangle\!\rangle^{pc} \approx_L \langle\!\langle c_2' \rangle\!\rangle^{pc}$ by Theorem 2 (λ^{dCG}-*TINI*). Finally, we deduce L-equivalence of the source final configurations again by the last point of Lemma 4.2, where $c_1' = \langle \Sigma, {r_1}^{\ell_1} \rangle$, $c_2' = \langle \Sigma_2, {r_2}^{\ell_2} \rangle$ and $pc \sqsubseteq \ell_1$ (resp. $pc \sqsubseteq \ell_2$) by Property 1 applied to the source reductions, i.e., $c_1 \Downarrow_{pc}^{\theta_1} c_1'$ (resp. $c_2 \Downarrow_{pc}^{\theta_2} c_2'$).

5 COARSE- TO FINE-GRAINED PROGRAM TRANSLATION

We now show a verified program translation in the opposite direction—from the coarse grained calculus λ^{dCG} to the fine grained calculus λ^{dFG}. The translation in this direction is more involved—a program in λ^{dFG} contains strictly more information than its counterpart in λ^{dCG}, namely the extra *intrinsic* label annotations that tag every value. The challenge in constructing this translation is two-fold. On one hand, the translation must come up with labels for all values. However, it is not always possible to do this statically during the translation: Often, the labels depend on input values and arise at run-time with intermediate results since the λ^{dFG} calculus is designed to compute and attach labels at run-time. On the other hand, the translation cannot conservatively under-approximate the values of

[16]Note that such bogus translations are also ruled out due to the need to preserve the outcome of any label introspection. Nonetheless, building this proof artifact increases our confidence in the robustness of our translation. In contrast, if the enforcement of IFC is *static*, then there is no label introspection, and this proof artifact is extremely important, as argued in prior work [Barthe et al. 2007; Rajani and Garg 2018].

labels[17]—λ^{dCG} and λ^{dFG} have label introspection so, in order to get semantics preservation, labels must be preserved precisely. Intuitively, we solve this impasse by crafting a program translation that (i) preserves the labels that can be inspected by λ^{dCG} and (ii) lets the λ^{dFG} semantics compute the remaining label annotations automatically—we account for those labels with a *cross-language* relation that represents semantic equivalence between λ^{dFG} and λ^{dCG} modulo extra annotations (Section 5.1). The fact that the source program in λ^{dCG} cannot inspect those labels—they have no value counterpart in the source λ^{dCG} program—facilitates this aspect of the translation. We elaborate more on the technical details later.

At a high level, an interesting aspect of the translation (that informally attests that it is indeed semantics-preserving) is that it encodes the *flow-sensitive* program counter of the source λ^{dCG} program into the label annotation of the λ^{dFG} value that results from executing the translated program. For example, if a λ^{dCG} monadic expression starts with program counter label pc and results in some value, say **true**, and final program counter pc', then the translated λ^{dFG} expression, starting with the same program counter label pc, computes the *same* value (modulo extra label annotations) at the same security level pc', i.e., the value $\mathbf{true}^{pc'}$. This encoding requires keeping the value of the program counter label in the source program synchronized with the program counter label in the target program, by loosening the fine-grained precision of λ^{dFG} at run-time in a controlled way.

Types. The λ^{dCG}-to-λ^{dFG} translation follows the same type-driven approach used in the other direction, starting from the function $[\![\cdot]\!]$ in Figure 14, that translates a λ^{dFG} type τ into the corresponding λ^{dCG} type $[\![\tau]\!]$. The translation is defined by induction on τ and preserves all the type constructors standard types. Only the cases corresponding to λ^{dCG}-specific types are interesting. In particular, it converts *explicitly* labeled types, i.e., **Labeled** τ, to a standard pair type in λ^{dFG}, i.e., $(\mathscr{L} \times [\![\tau]\!])$, where the first component is the label and the second component the content of type τ. Type **LIO** τ becomes a *suspension* in λ^{dFG}, i.e., the function type $\mathbf{unit} \rightarrow [\![\tau]\!]$ that delays a computation and that can be forced by simply applying it to the unit value ().

Values. The translation of values follows the type translation, as shown in Figure 15. Notice that the translation is indexed by the program counter label (the translation is written $[\![v]\!]^{pc}$), which converts the λ^{dCG} value v in scope of a computation protected by security level pc to the corresponding fully label-annotated λ^{dFG} value. The translation is pretty straightforward and uses the program counter label to tag each value, following the λ^{dCG} principle that the program counter label protects every value in scope that is not explicitly labeled. The translation converts a λ^{dCG} function closure into a corresponding λ^{dFG} function closure by translating the body of the function to a λ^{dFG} expression (see below) and translating the environment pointwise, i.e., $[\![\theta]\!]^{pc} = \lambda x.[\![\theta(x)]\!]^{pc}$. A thunk value or a *thunk closure*, which denotes a suspended side-effecful computation, is also converted into a λ^{dFG} function closure. Technically, the translation would need to introduce a *fresh variable* that would get bound to unit when the suspension gets forced. However, the argument to the suspension does not have any purpose, so we do not bother with giving a name to it and write $\_.[\![t]\!]$ instead. (In our mechanized proofs we employ unnamed De Bruijn indexes and this issue does not arise.) The translation converts an explicitly labeled value **Labeled** ℓ v,

[17]In contrast, previous work on *static* type-based fine-to-coarse grained translation safely under-approximates the label annotations in types with $\bot$ [Rajani and Garg 2018]. The proof of type preservation of the translation recovers the actual labels via *subtyping*.

into a labeled pair at security level pc, i.e., $(\ell^{\ell}, [\![v]\!]^{\ell})^{pc}$. The pair consists of the label ℓ tagged with itself, and the value translated at a security level equal to the label annotation, i.e., $[\![v]\!]^{\ell}$. Notice that tagging the label with itself allows us to translate the λ^{dCG} (label introspection) primitive $\mathbf{labelOf}(\cdot)$ by simply projecting the first component, thus preserving the label and its security level across the translation.

Expressions and Thunks. The translation of pure expressions (Figure 16) is trivial: It is homomorphic in all constructs, mirroring the type translation. The translation of a thunk expression t builds a suspension explicitly with a λ-abstraction (the name of the variable is again irrelevant, thus we omit it as explained above), and carries on by translating the thunk itself according to the definition in Figure 17. The thunk $\mathbf{return}(e)$ becomes $[\![e]\!]$, since $\mathbf{return}(\cdot)$ does not have any side-effect. When two monadic computations are combined via $\mathbf{bind}(e_1, x.e_2)$, the translation (i) converts the first computation to a suspension and forces it by applying unit ($[\![e_1]\!]()$), (ii) binds the result to x and passes it to the second computation[18], which is also converted, forced, and, *importantly*, iii) executed with a program counter label tainted with the security level of the result of the first computation ($\mathbf{taint}(\mathbf{labelOf}(x), [\![e_2]\!]())$). Notice that $\mathbf{taint}(\cdot)$ is essential to ensure that the second computation executes with the program counter label set to the correct value—the precision of the fine-grained system would otherwise retain the initial lower program counter label according to rule [App] and the value of the program counter labels in the source and target programs would differ in the remaining execution.

Similarly, the translation of $\mathbf{unlabel}(e)$ first translates the labeled expression e (the translated expression does not need to be forced because it is not of a monadic type), binds its result to x and then projects the content in a context tainted with its label, as in $\mathbf{taint}(\mathbf{fst}(x), \mathbf{snd}(x))$. This closely follows λ^{dCG}'s [Unlabel] rule. The translation of $\mathbf{toLabeled}(e)$ forces the nested computation with $[\![e]\!]()$, binds its result to x and creates the pair $(\mathbf{labelOf}(x), x)$, which corresponds to the labeled value obtained in λ^{dCG} via rule [ToLabeled]. Intuitively, the translation guarantees that the value of the final program counter label in the nested computation coincides with the security level of the translated result (bound to x). Therefore, the first component contains the correct label and it is furthermore at the right security level, because $\mathbf{labelOf}(\cdot)$ protects the projected label with the label itself in λ^{dFG}. Primitive $\mathbf{labelOf}(e)$ simply projects the first component of the pair that encodes the labeled value in λ^{dFG} as explained above. Lastly, $\mathbf{getLabel}$ in λ^{dCG} maps directly to $\mathbf{getLabel}$ in λ^{dFG}—rule [GetLabel] in λ^{dCG} simply returns the program counter label and does not raise its value, so it corresponds exactly to rule [GetLabel] in λ^{dFG}, which returns label pc at security level pc. Similarly, $\mathbf{taint}(e)$ translates to $\mathbf{taint}([\![e]\!], ())$ since rule [Taint] in λ^{dCG} simply taints the program counter label with the label that e evaluates to, say ℓ and results in unit with program counter label raised to $pc \sqcup \ell$. This corresponds to the result of the translated program, i.e., $()^{pc \sqcup \ell}$.

References. Figure 18 shows the translation of primitives that access the store via references. Since λ^{dCG}'s rule [New] in Figure 8 creates a new reference labeled with the label of the argument (which must be a labeled value), the translation converts $\mathbf{new}(e)$ to an expression that first binds $[\![e]\!]$ to x and then creates a new reference with the same content as the source, i.e., $\mathbf{snd}(x)$, but tainted with the label in x, i.e., $\mathbf{fst}(x)$. Notice that the use of $\mathbf{taint}(\cdot)$ is essential to ensure that λ^{dFG}'s rule [New] in Figure 4 assigns the correct label to the new reference. Due to its *fine-grained* precision, λ^{dFG} might have labeled the content with a

[18]Syntax $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ where x is free in e_2 is a shorthand for $(\lambda x.e_2)\ e_1$.

different label that is less sensitive than the explicit label that *coarsely* approximates the security level in λ^{dCG}. In contrast, updating a reference does not require any tainting—both λ^{dFG} and λ^{dCG} accept values less sensitive than the reference in rule [Write]. Thus, the translation $e_1 := e_2$ simply updates the translated reference with the content of the labeled value projected from the translated pair. Hence, $[\![e_1 := e_2]\!]$ is $[\![e_1]\!] := \mathbf{snd}([\![e_2]\!])$. The translation of the primitives that read and query the label of a reference is trivial.

5.1 Cross-Language Semantic Equivalence up to Extra Annotations

When a λ^{dCG} program is translated to λ^{dFG} via the program translation described above, the λ^{dFG} result contains strictly more information than the original λ^{dCG} result. This happens because the semantics of λ^{dFG} tracks flows of information at fine granularity, in contrast with λ^{dCG}, which instead coarsely approximates the security level of all values in scope of a computation with the program counter label. When translating a λ^{dCG} program, we can capture this condition precisely for input values θ by *homogeneously* tagging all standard (unlabeled) values with the initial program counter label, i.e., $[\![\theta]\!]^{pc}$. However, a λ^{dCG} program handles, creates and mixes unlabeled data that originated at different security levels at run-time, e.g., when a secret is unlabeled and combined with previously public (unlabeled) data. Crucially, when the translated program executes, the fine-grained semantics of λ^{dFG} tracks those flows of information precisely and thus new labels appear (these labels do not correspond to the label of any labeled value in the source value nor to the program counter label). Intuitively, this implies that the λ^{dFG} result will *not* be homogeneously labeled with the final program counter label of the λ^{dCG} computation, i.e., if a λ^{dCG} program terminates with value v and program counter label pc', the translated λ^{dFG} program does not necessarily result in $[\![v]\!]^{pc'}$.

Example. Consider the λ^{dCG} program $\langle \Sigma, L, \mathbf{taint}(H); \mathbf{return}(x)\rangle \Downarrow^{x \mapsto \mathbf{true}} \langle \Sigma, H, \mathbf{true}\rangle$, which returns $\mathbf{true} = \mathbf{inl}()$ and the store Σ unchanged, after tainting the program counter label with H. Let e be the expression obtained by applying the program translation defined above to the example program:

$$\begin{aligned} e = {} & \lambda\_. \\ & \mathbf{let}\ y = \mathbf{taint}(H, ())\ \mathbf{in} \\ & \quad \mathbf{taint}(\mathbf{labelOf}(y), x) \end{aligned}$$

Interestingly, when we force the program e and execute it starting from program counter label equal to L, and an input environment translated according to the initial program counter label (L in this case), i.e., $x \mapsto [\![\mathbf{true}]\!]^L = \mathbf{inl}(()^L)^L = \mathbf{true}^L$, we do *not* obtain the translated result homogeneously labeled with H:

$$\langle [\![\Sigma]\!], e\ ()\rangle \Downarrow_L^{x \mapsto \mathbf{true}^L} \langle [\![\Sigma]\!], \mathbf{true}^H\rangle = \langle [\![\Sigma]\!], \mathbf{inl}(()^L)^H\rangle \neq \langle [\![\Sigma]\!], \mathbf{inl}(()^H)^H\rangle = \langle [\![\Sigma]\!], [\![\mathbf{true}]\!]^H\rangle$$

In particular, λ^{dFG} preserves the public label tag on data nested inside the left injection, i.e., $()^L$ in $\mathbf{inl}(()^L)^H$ above. This happens because λ^{dFG}'s rule [Var] taints only the *outer* label of the value $\mathbf{true}^L$ when it looks up variable x in program counter label H.

Solution. In order to recover a notion of semantics preservation, we introduce a key contribution of this work, a *cross-language* binary relation that associates values of the two calculi that, in the scope of a computation at a given security level, are semantically equivalent up to the extra annotations present in the λ^{dFG} value.[19] Technically, we use this equivalence in the semantics preservation theorem in Section 5.2, which *existentially* quantifies over the result of the translated λ^{dFG} program, but guarantees that it is semantically equivalent to the result obtained in the source program.

Concretely, for a λ^{dFG} value v_1 and a λ^{dCG} value v_2, we write $v_1 \downarrow\approx_{pc} v_2$ if the label annotations (including those nested inside compound values) of v_1 are no more sensitive than label pc and its raw value corresponds to v_2. Figure 19 formalizes this intuition by means of two mutually inductive relations, one for λ^{dFG} values and one for λ^{dFG} raw values. In particular, rule [VALUE] relates λ^{dFG} value ${r_1}^{\ell_1}$ and λ^{dCG} value v_2 at security level pc if the label annotation on the raw value r_1 flows to the program counter label, i.e., $\ell_1 \sqsubseteq pc$, and if the raw value is in relation with the standard value, i.e., $r_1 \downarrow\approx_{pc} v_2$. The relation between raw values and standard values relates only semantically equivalent values, i.e., it is syntactic equality for ground types ([UNIT,LABEL,REF]), requires the same injection for values of the sum type ([INL,INR]) and requires the components to related for pairs ([PAIR]).

Rules [FUN] (resp. [THUNK]) relates function (resp. thunk) closures only when environments are related pointwise, i.e., $\theta_1 \downarrow\approx_{pc} \theta_2$ iff $Dom(\theta_1) \equiv Dom(\theta_2)$ and $\forall x.\theta_1(x) \downarrow\approx_{pc} \theta_2(x)$, and the λ^{dFG} function body $x.[\![e]\!]$ (resp. thunk body $\_.[\![t]\!]$) is obtained from the λ^{dCG} function body e (resp. thunk t) via the program translation defined above. Lastly, rule [LABELED] relates a λ^{dCG} labeled value **Labeled** $\ell\ v_1$ to a pair (ℓ^{ℓ}, v_2), consisting of the label ℓ protected by itself in the first component and value v_2 related with the content v_1 at security level ℓ ($v_1 \downarrow\approx_{\ell} v_2$) in the second component. This rule follows the principle of **LIO** that for explicitly labeled values, the label annotation represents an upper bound on the sensitivity of the content. Stores are related pointwise, i.e., $\Sigma_1 \downarrow\approx \Sigma_2$ iff $\Sigma_1(\ell) \downarrow\approx \Sigma_2(\ell)$ for $\ell \in \mathscr{L}$, and ℓ-labeled memories relate their contents respectively at security level ℓ, i.e., $[\,] \downarrow\approx [\,]$ and $(r_1 : M_1) \downarrow\approx (r_2 : M_2)$ iff $r_1 \downarrow\approx_{\ell} r_2$ and $M_1 \downarrow\approx M_2$ for λ^{dFG} and λ^{dCG} memories $M_1, M_2 : Memory\ \ell$. Lastly, we lift the relation to initial and final configurations.

DEFINITION 1 (EQUIVALENCE OF CONFIGURATIONS). *For all initial and final configurations:*

- $\langle \Sigma_1, [\![e]\!]() \rangle \downarrow\approx \langle \Sigma_2, pc, e \rangle$ *iff* $\Sigma_1 \downarrow\approx \Sigma_2$,
- $\langle \Sigma_1, [\![t]\!] \rangle \downarrow\approx \langle \Sigma_2, pc, t \rangle$ *iff* $\Sigma_1 \downarrow\approx \Sigma_2$,
- $\langle \Sigma_1, r^{pc} \rangle \downarrow\approx \langle \Sigma_2, pc, v \rangle$ *iff* $\Sigma_1 \downarrow\approx \Sigma_2$ *and* $r \downarrow\approx_{pc} v$.

For initial configurations, the relation requires the λ^{dFG} code to be obtained from the λ^{dCG} expression (resp. thunk) via the program translation function $[\![\cdot]\!]$ defined above (similar to rules [FUN] and [THUNK] in Figure 19). Furthermore, in the first case (expressions), the relation additionally forces the translated suspension $[\![e]\!]$ by applying it to (), so that when the λ^{dFG} security monitor executes the translated program, it obtains the result that corresponds to the λ^{dCG} monadic program e. The third definition relates final configurations whenever the stores are related and the security level of the final λ^{dFG} result corresponds to the program counter label pc of the final λ^{dCG} configuration, and the final λ^{dCG} result corresponds to the λ^{dFG} result up to extra annotations at security level pc, i.e., $r \downarrow\approx_{pc} v$.

Before showing semantics preservation, we prove some basic properties of the equivalence that will be useful later. The following property allows instantiating the semantics preservation theorem with the λ^{dCG} initial configuration. The translation for initial configurations is per-component, i.e., $[\![\langle \Sigma, pc, t \rangle]\!] = \langle [\![\Sigma]\!], [\![t]\!] \rangle$ and forcing for suspensions, i.e.,

[19] This relation is conceptually similar to the logical relation developed by Rajani and Garg [2018] for their translations with *static* IFC enforcement, but technically different in the treatment of labeled values.

$[\![\langle \Sigma, pc, e \rangle]\!] = \langle [\![\Sigma]\!], [\![e]\!]() \rangle$, pointwise for stores, i.e., $[\![\Sigma]\!] = \lambda \ell. [\![\Sigma(\ell)]\!]$, and memories, i.e., $[\![[\,]]\!] = [\,]$ and $[\![v : M]\!] = [\![v]\!]^{\ell} : [\![M]\!]$ for ℓ-labeled memory M.

Property 3 (Reflexivity). *For all initial configurations c, $[\![c]\!] \downarrow\approx c$.*

Proof. The proof is by induction and relies on analogous properties for all syntactic categories: for stores, $[\![\Sigma]\!] \downarrow\approx \Sigma$, for memories, $[\![M]\!] \downarrow\approx M$, for values $[\![v]\!]^{pc} \downarrow\approx_{pc} v$, for environments $[\![\theta]\!]^{pc} \downarrow\approx_{pc} \theta$, for any label pc.

The next property guarantees that values and environments remain in the relation when the program counter label rises.

Property 4 (Weakening). *For all labels pc and pc' such that $pc \sqsubseteq pc'$, and for all λ^{dFG} values v_1 and environments θ_1, and λ^{dCG} values v_2 and environments θ_2:*

- *If $v_1 \downarrow\approx_{pc} v_2$ then $v_1 \downarrow\approx_{pc'} v_2$*
- *If $\theta_1 \downarrow\approx_{pc} \theta_2$ then $\theta_1 \downarrow\approx_{pc'} \theta_2$*

Proof. By mutual induction on the relation and using basic properties of the lattice.

5.2 Correctness

With the help of the cross-language relation defined above, we can now state and prove that the λ^{dCG}-to-λ^{dFG} translation is correct, i.e., it satisfies a semantics-preservation theorem analogous to that proved for the translation in the opposite direction. At a high level, the theorem ensures that the translation preserves the meaning of a secure terminating λ^{dCG} program when executed under λ^{dFG} semantics, with the same program counter label and translated input values. Since the translated λ^{dFG} program computes strictly more information than the original λ^{dCG} program, the theorem existentially quantify over the λ^{dFG} result, but insists that it is semantically equivalent to the original λ^{dCG} result at a security level equal to the final value of the program counter label, using the cross-language relation just defined.

We start by proving that the program translation preserves typing.

Lemma 5.1 (Type Preservation). *If $\Gamma \vdash e : \tau$ then $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$.*

Proof. By straightforward induction on the typing judgment.

Next, we prove semantics preservation of λ^{dCG} pure reductions. Since these reductions do not perform any security-relevant operation (they do not read or write state), they can be executed with *any* program counter label in λ^{dFG} and do not change the state in λ^{dFG}.

Lemma 5.2 ($[\![\cdot]\!] : \lambda^{dCG} \to \lambda^{dFG}$ preserves Pure Semantics). *If $e \Downarrow^{\theta} v$ then for any program counter label pc, λ^{dFG} store Σ, environment θ' such that $\theta' \downarrow\approx_{pc} \theta$, there exists a raw value r, such that $\langle \Sigma, [\![e]\!] \rangle \Downarrow^{\theta'}_{pc} \langle \Sigma, r^{pc} \rangle$ and $r \downarrow\approx_{pc} v$.*

Proof. By induction on the given evaluation derivation and using basic properties of the lattice.

Notice that the lemma holds for *any* input target environment θ' in relation with the source environment θ at security level pc rather than just for the translated environment $[\![\theta]\!]^{pc}$. Intuitively, we needed to generalize the lemma so that the induction principle is strong enough to discharge cases where (i) we need to prove reductions that use an existentially quantified environment, e.g., [App] and (ii) when some intermediate result at a security level other than pc gets added to the environment, so the environment is no longer homogenously labeled with pc. While the second condition does not arise in pure reductions, it does occur in the reduction of monadic expressions considered in the following theorem.

Theorem 5 ($[\![\cdot]\!] : \lambda^{dCG} \to \lambda^{dFG}$ preserves Thunk and Forcing Semantics).

- *Let $c_2 = \langle \Sigma_2, pc, t \rangle$ be an initial λ^{dCG} configuration. If $c_2 \Downarrow^{\theta_2} c_2'$, then for all λ^{dFG} environments θ_1 and initial configurations c_1 such that $\theta_1 \downarrow\approx_{pc} \theta_2$ and $c_1 \downarrow\approx c_2$, there exists a final configuration c_1', such that $c_1 \Downarrow_{pc}^{\theta_1} c_1'$ and $c_1' \downarrow\approx c_2'$.*
- *Let $c_2 = \langle \Sigma_2, pc, e \rangle$ be an initial λ^{dCG} configuration. If $c_2 \Downarrow^{\theta_2} c_2'$, then for all λ^{dFG} environments θ_1 and initial configurations c_1 such that $\theta_1 \downarrow\approx_{pc} \theta_2$ and $c_1 \downarrow\approx c_2$, there exists a final configuration c_1', such that $c_1 \Downarrow_{pc}^{\theta_1} c_1'$ and $c_1' \downarrow\approx c_2'$.*

Proof (Sketch). By mutual induction on the given derivations, using Lemma 5.2 for pure reductions and Properties 2 and 4 in cases [Bind, ToLabeled, Unlabel, Read], basic properties of the lattice and of the translation function (for operations on the store).

We finally instantiate the semantics-preservation theorem with the translation of the input values and the initial stores at security level pc.

Corollary 1 (Correctness). *Let $c_1 = \langle \Sigma, pc, e \rangle$, if $c_1 \Downarrow^{\theta} c_1'$, then there exists a final λ^{dFG} configuration c_2' such that $[\![c_1]\!] \Downarrow_{pc}^{[\![\theta]\!]^{pc}} c_2'$ and $c_1' \downarrow\approx c_2'$.*

Proof. By Property 3 and Theorem 5.

Notice that the flow-sensitive program counter of the source λ^{dCG} program gets encoded in the security level of the result of the λ^{dFG} translated program. For example, if $\langle \Sigma_2, pc, e \rangle \Downarrow^{\theta} \langle \Sigma_2', pc', v \rangle$ then, by Corollary 1 and unrolling Definition 1, there exists a raw value r at security level pc' and a store Σ_1', such that $\langle [\![\Sigma_2]\!], [\![e]\!]() \rangle \Downarrow_{pc}^{[\![\theta]\!]^{pc}} \langle \Sigma_1', r^{pc'} \rangle$, $r \downarrow\approx_{pc'} v$ and $\Sigma_1' \downarrow\approx \Sigma_2'$.

Recovery of non-interference. Similarly to our presentation of Theorem 4 for the translation in the opposite direction, we conclude this section with a sanity check—recovering a proof of termination-insensitive non-interference (*TINI*) for λ^{dCG} through the program translation defined above, semantics preservation and the non-interference of λ^{dFG}. By reproving non-interference of the source language from the target language, we show that our program translation is authentic.

The following lemma ensures that the translation of initial configurations preserves L-equivalence.

Lemma 5.3. *If $c_1 \approx_L c_2$, then $[\![c_1]\!] \approx_L [\![c_2]\!]$.*

Proof. By induction on the L-equivalence judgment and proving similar lemmas for values, i.e., if $v_1 \approx_L v_2$ then $[\![v_1]\!]^{pc} \approx_L [\![v_2]\!]^{pc}$, for environments, i.e., if $\theta_1 \approx_L \theta_2$ then $[\![\theta_1]\!]^{pc} \approx_L [\![\theta_2]\!]^{pc}$, for any label pc, for memories, i.e., if $M_1 \approx_L M_2$ then $[\![M_1]\!] \approx_L [\![M_2]\!]$, and for stores, i.e., if $\Sigma_1 \approx_L \Sigma_2$ then $[\![\Sigma_1]\!] \approx_L [\![\Sigma_2]\!]$.

The following lemmas recovers λ^{dCG} L-equivalence from λ^{dFG} L-equivalence using the cross-language equivalence relation for all the syntactic categories.

Lemma 5.4. *For all public program counter labels $pc \sqsubseteq L$, for all λ^{dFG} values v_1, v_2, raw values r_1, r_2, environments θ_1, θ_2, memories M_1, M_2, stores Σ_1, Σ_2, and corresponding λ^{dCG} values v_1', v_2' and environments θ_1', θ_2', memories M_1', M_2', stores Σ_1', Σ_2':*

- *If $v_1 \approx_L v_2$, $v_1 \downarrow\approx_{pc} v_1'$ and $v_2 \downarrow\approx_{pc} v_2'$, then $v_1' \approx_L v_2'$,*
- *If $r_1 \approx_L r_2$, $r_1 \downarrow\approx_{pc} v_1'$ and $r_2 \downarrow\approx_{pc} v_2'$, then $v_1' \approx_L v_2'$,*
- *If $\theta_1 \approx_L \theta_2$, $\theta_1 \downarrow\approx_{pc} \theta_1'$ and $\theta_2 \downarrow\approx_{pc} \theta_2'$, then $\theta_1' \approx_L \theta_2'$,*
- *If $M_1 \approx_L M_2$, $M_1 \downarrow\approx M_1'$ and $M_2 \downarrow\approx M_2'$, then $M_1' \approx_L M_2'$,*
- *If $\Sigma_1 \approx_L \Sigma_2$, $\Sigma_1 \downarrow\approx \Sigma_1'$ and $\Sigma_2 \downarrow\approx \Sigma_2'$, then $\Sigma_1' \approx_L \Sigma_2'$.*

Proof. The lemmas are proved mutually, by induction on the L-equivalence relation and the cross-language equivalence relations and using injectivity of the translation function $[\![\cdot]\!]$ for closure values.[20]

The next lemma lifts the previous lemma final configurations.

LEMMA 5.5. *Let c_1 and c_2 be λ^{dFG} final configurations, let c_1' and c_2' be λ^{dCG} final configurations. If $c_1 \approx_L c_2$, $c_1 \downarrow\approx c_1'$ and $c_2 \downarrow\approx c_2'$, then $c_1' \approx_L c_2'$.*

Proof. Let $c_1 = \langle \Sigma_1, v_1 \rangle$, $c_2 = \langle \Sigma_2, v_2 \rangle$, $c_1' = \langle \Sigma_1', pc_1, v_1' \rangle$, $c_2' = \langle \Sigma_2', pc_2, v_2' \rangle$. From L-equivalence of λ^{dFG} final configurations, it follows L-equivalence for the stores and the values, i.e., $\Sigma_1 \approx_L \Sigma_2$ and $v_1 \approx_L v_2$ from $c_1 \approx_L c_2$ (Section 2.2). Similarly, from cross-language equivalence of final λ^{dFG} and λ^{dCG} configurations, it follows cross-language equivalence of their components, i.e., respectively $\Sigma_1 \downarrow\approx \Sigma_1'$ and $v_1 \downarrow\approx_{pc_1} v_1'$ from $c_1 \downarrow\approx c_2$, and $\Sigma_2 \downarrow\approx \Sigma_2'$ and $v_2 \downarrow\approx_{pc_2} v_2'$ from $c_2 \downarrow\approx c_2'$ (Definition 1). First, we show that the λ^{dCG} stores are L-equivalent, i.e., $\Sigma_1' \approx_L \Sigma_2'$ by Lemma 5.4 for stores, then two cases follow by case split on $v_1 \approx_L v_2$. Either (i) both label annotations on the values are not observable ([$\textsc{Value}_H$]), then the program counter labels are also not observable ($pc_1 \not\sqsubseteq L$ and $pc_2 \not\sqsubseteq L$ from $v_1 \downarrow\approx_{pc_1} v_1'$ and $v_2 \downarrow\approx_{pc_2} v_2'$) and $c_1' \approx_L c_2'$ by rule [$\textsc{Pc}_H$] or (ii) the label annotations are equal and observable by the attacker ([$\textsc{Value}_L$]), i.e., $pc_1 = pc_2 \sqsubseteq L$, then $v_1' \approx_L v_2'$ by Lemma 5.4 for values and $c_1' \approx_L c_2'$ by rule [$\textsc{Pc}_L$].

THEOREM 6 (λ^{dCG}-TINI VIA $[\![\cdot]\!]$). *If $c_1 \Downarrow^{\theta_1} c_1'$, $c_2 \Downarrow^{\theta_2} c_2'$, $\theta_1 \approx_L \theta_2$ and $c_1 \approx_L c_2$, then $c_1' \approx_L c_2'$.*

Proof. First, we apply the translation $[\![\cdot]\!] : \lambda^{dCG} \to \lambda^{dFG}$ to the initial configurations c_1 and c_2 and the respective environments θ_1 and θ_2. Let pc be the initial program counter label common to configurations c_1 and c_2 (it is the same because $c_1 \approx_L c_2$). Corollary 1 (Correctness) then ensures that there exist two λ^{dFG} configurations c_1'' and c_2'', such that $[\![c_1]\!] \Downarrow_{pc}^{[\![\theta_1]\!]^{pc}} c_1''$ and $c_1'' \downarrow\approx c_1'$, and $[\![c_2]\!] \Downarrow_{pc}^{[\![\theta_2]\!]^{pc}} c_2''$ and $c_2'' \downarrow\approx c_2'$. We then lift L-equivalence of source configurations and environments to L-equivalence in the target language via Lemma 5.3, i.e., $[\![\theta_1]\!]^{pc} \approx_L [\![\theta_2]\!]^{pc}$ and $[\![c_1]\!] \approx_L [\![c_2]\!]$, and apply Theorem 1 ($\lambda^{dFG}$-*TINI*) to the reductions i.e., $[\![c_1]\!] \Downarrow_{pc}^{[\![\theta_1]\!]^{pc}} c_1''$ and $[\![c_2]\!] \Downarrow_{pc}^{[\![\theta_2]\!]^{pc}} c_2''$, which gives L-equivalence of the resulting configurations, i.e., $c_1'' \approx_L c_2''$. Then, we apply Lemma 5.5 to $c_1'' \approx_L c_2''$, $c_1'' \downarrow\approx c_1'$, and $c_2'' \downarrow\approx c_2'$, and recover L-equivalence for the source configurations, i.e., $c_1' \approx_L c_2'$.

6 RELATED WORK

Systematic study of the relative expressiveness of fine- and coarse-grained information flow control (IFC) systems has started only recently. Rajani et al. [2017] initiated this study in the context of *static* coarse- and fine-grained IFC, enforced via type systems. In more recent work, Rajani and Garg [2018] show that a fine-grained IFC type system, which they call FG, and two variants of a coarse-grained IFC type system, which they call CG, are equally expressive. Their approach is based on type-directed translations, which are type- and semantics-preserving. For proofs, they develop logical relations models of FG and the

[20]Technically, the function $[\![\cdot]\!]$ presented in Section 5 is not injective. For example, consider the type translation function from Figure 14: $[\![\textbf{Labeled unit}]\!] = \mathscr{L} \times \textbf{unit} = [\![\mathscr{L} \times \textbf{unit}]\!]$ but $\textbf{Labeled unit} \neq \mathscr{L} \times \textbf{unit}$, and $[\![\textbf{LIO unit}]\!] = \textbf{unit} \to \textbf{unit} = [\![\textbf{unit} \to \textbf{unit}]\!]$ but $\textbf{LIO unit} \neq \textbf{unit} \to \textbf{unit}$. We make the translation injective by (i) adding a wrapper type $\textbf{Id}\ \tau$ to λ^{dFG}, together with constructor $\textbf{Id}(e)$, a deconstructor $\textbf{unId}(e)$ and raw value $\textbf{Id}(v)$, and (ii) tagging security-relevant types and terms with the wrapper, i.e., $[\![\textbf{Labeled}\ \tau]\!] = \textbf{Id}\ (\mathscr{L} \times [\![\tau]\!])$ and $\textbf{LIO}\ \tau = \textbf{Id}\ (\textbf{unit} \to [\![\tau]\!])$. Adapting the translations in both directions is tedious but straightforward; we refer the interested reader to our mechanized proofs for details.

two variants of CG, as well as cross-language logical relations. Our work and some of our techniques are directly inspired by their work, but we examine *dynamic* IFC systems based on runtime monitors. As a result, our technical development is completely different. In particular, in our work we handle label introspection, which has no counterpart in the earlier work on static IFC systems, and which also requires significant care in translations. Our dynamic setting also necessitated the use of tainting operators in both the fine-grained and the coarse-grained systems.

Our coarse-grained system λ^{dCG} is the dynamic analogue of the second variant of Rajani and Garg [2018]'s CG type system. This variant is described only briefly in their paper (in Section 4, paragraph "Original HLIO") but covered extensively in Part-II of the paper's appendix. Rajani and Garg [2018] argue that translating their fine-grained system FG to this variant of CG is very difficult and requires significant use of parametric label polymorphism. The astute reader may wonder why we do not encounter the same difficulty in translating our fine-grained system λ^{dFG} to λ^{dCG}. The reason for this is that our fine-grained system λ^{dFG} is not a direct dynamic analogue of Rajani and Garg [2018]'s FG. In λ^{dFG}, a value constructed in a context with program counter label *pc* automatically receives the security label *pc*. In contrast, in Rajani and Garg [2018]'s FG, all introduction rules create values (statically) labeled $\perp$. Hence, leaving aside the static-vs-dynamic difference, FG's labels are more precise than λ^{dFG}'s, and this makes Rajani and Garg [2018]'s FG to CG translation more difficult than our λ^{dFG} to λ^{dCG} translation. In fact, in earlier work, Rajani et al. [2017] introduced a different type system called FG^{-}, a static analogue of λ^{dFG} that labels all constructed values with *pc* (statically), and noted that translating it to the second variant of CG is much easier (in the static setting).

Coarse-grained dynamic IFC systems are prevalent in security research in operating systems [Efstathopoulos et al. 2005; Krohn et al. 2007a; Zeldovich et al. 2006]. These ideas have also been successfully applied to other domains, e.g., the web [Bauer et al. 2015; Giffin et al. 2012; Stefan et al. 2014; Yip et al. 2009], mobile applications [Jia et al. 2013; Nadkarni et al. 2016], and IoT [Fernandes et al. 2016]. LIO is a domain-specific language embedded in Haskell that rephrases OS-like IFC enforcement into a language-based setting [Stefan et al. 2012, 2011]. Heule et al. [2015] introduce a general framework for coarse-grained IFC in any programming language in which external effects can be controlled. Laminar [Roy et al. 2009] unifies mechanisms for IFC in programming languages and operating systems, resulting in a mix of dynamic fine- and coarse-grained enforcement.

In general, dynamic fine-grained IFC systems often do not support label introspection. LIO [Stefan et al. 2017, 2011] and Breeze [Hritcu et al. 2013b] are notable exceptions. Breeze is conceptually similar to our λ^{dFG} except for the $\textbf{taint}(\cdot)$ primitive. Different from our λ^{dFG}, there are dynamic fine-grained IFC systems in which labels of references are flow-sensitive [Austin and Flanagan 2009, 2010; Bichhawat et al. 2014; Hedin et al. 2014]. This design choice, however, allows label changes to be exploited as a covert channel for information leaks [Austin and Flanagan 2009, 2010; Russo and Sabelfeld 2010]. There are many approaches to preventing such leaks—from using static analysis techniques [Sabelfeld and Myers 2003], to disallowing label upgrades depending on sensitive data (i.e., no-sensitive-upgrades [Austin and Flanagan 2009; Zdancewic 2002]), to avoiding branching on data whose labels have been upgraded (i.e., permissive-upgrades [Austin and Flanagan 2010]). Extending our results to a fine-grained dynamic IFC system with flow-sensitive references is an interesting direction for future work.

7 CONCLUSION

We formally established a connection between dynamic fine- and coarse-grained enforcement for IFC, showing that both are equally expressive under reasonable assumptions. Indeed, this work provides a systematic way to bridging the gap between a wide range of dynamic IFC techniques often proposed by the programming languages (fine-grained) and operating systems (coarse-grained) communities. As consequence, this allows future designs of dynamic IFC to choose a coarse-grained approach, which is easier to implement and use, without giving up on the precision of fine-grained IFC.

ACKNOWLEDGMENTS

We thanks the anonymous POPL and POPL AEC reviewers for the insightful comments. This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011) as well as the Swedish research agency Vetenskapsrådet. Vineet Rajani was partly funded through the Collaborative Research Center "Methods and Tools for Understanding and Controlling Privacy" (SFB 1223) of the DFG, project "Programming Principles and Abstractions for Privacy." This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

T. H. Austin and C. Flanagan. 2009. Efficient Purely-Dynamic Information Flow Analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*.

Thomas H. Austin and Cormac Flanagan. 2010. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '10)*. ACM.

Gilles Barthe, Tamara Rezk, and Amitabh Basu. 2007. Security types preserving compilation. *Computer Languages, Systems & Structures* 33, 2 (2007), 35–59. https://doi.org/10.1016/j.cl.2005.05.002

Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time monitoring and formal analysis of information flows in Chromium. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium*. Internet Society.

Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Information Flow Control in WebKit's JavaScript Bytecode. In *International Conference on Principles of Security and Trust (POST)*. 159–178.

Niklas Broberg, Bart van Delft, and David Sands. 2013. Paragon for Practical Programming with Information-Flow Control. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*. 217–232.

Pablo Buiras, Deian Stefan, and Alejandro Russo. 2014. On Dynamic Flow-Sensitive Floating-Label Systems. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium (CSF '14)*. IEEE Computer Society, Washington, DC, USA, 65–79. https://doi.org/10.1109/CSF.2014.13

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 280–288.

Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. 2005. Labels and event processes in the asbestos operating system. In *Proc. of the twentieth ACM symp. on Operating systems principles (SOSP '05)*. ACM.

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (Dec. 1991), 35–75. https://doi.org/10.1016/0167-6423(91)90036-W

Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks.. In *USENIX*

Security Symposium. 531–548.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. 2012. Hails: Protecting Data Privacy in Untrusted Web Applications. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI*.

J.A. Goguen and J. Meseguer. 1982. Security policies and security models. In *Proc of IEEE Symposium on Security and Privacy*. IEEE Computer Society.

D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. of the ACM Symposium on Applied Computing (SAC '14)*. ACM.

Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. 2015. IFC Inside: Retrofitting Languages with Dynamic Information Flow Control. In *Proceedings of the Conference on Principles of Security and Trust*. Springer.

C. Hritcu, M. Greenberg, B. Karel, B. C. Peirce, and G. Morrisett. 2013a. All Your IFCException Are Belong to Us. In *Proc. of the IEEE Symposium on Security and Privacy*. IEEE Computer Society.

Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C Pierce, and Greg Morrisett. 2013b. All your IFCException are belong to us. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 3–17.

M. Jaskelioff and A. Russo. 2011. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (LNCS)*. Springer-Verlag.

Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-Time Enforcement of Information-Flow Properties on Android (Extended Abstract). In *Computer Security—ESORICS 2013: 18th European Symposium on Research in Computer Security*. Springer.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007a. Information Flow Control for Standard OS Abstractions. In *Proc. of the 21st Symp. on Operating Systems Principles*.

Maxwell N. Krohn, Alexander Yip, Micah Z. Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Tappan Morris. 2007b. Information flow control for standard OS abstractions. In *Proc. of the 21st ACM Symposium on Operating Systems Principles*. 321–334.

Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. Jif 3.0: Java information flow. (July 2006). http://www.cs.cornell.edu/jif

Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC Enforcement on Android.. In *USENIX Security Symposium*. 1119–1136.

F. Pottier and V. Simonet. 2002. Information Flow Inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*. 319–330.

Vineet Rajani, Iulia Bastys, Willard Rafnsson, and Deepak Garg. 2017. Type systems for information flow control: the question of granularity. *SIGLOG News* 4, 1 (2017), 6–21.

Vineet Rajani and Deepak Garg. 2018. Types for Information Flow Control: Labeling Granularity and Semantic Models. In *Proc. of the IEEE Computer Security Foundations Symp. (CSF '18)*. IEEE Computer Society.

Indrajit Roy, Donald E Porter, Michael D Bond, Kathryn S McKinley, and Emmett Witchel. 2009. *Laminar: Practical fine-grained decentralized information flow control*. Vol. 44. ACM.

Alejandro Russo. 2015. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proc. of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM.

A. Russo, K. Claessen, and J. Hughes. 2008. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM.

Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proc. of the 2010 23rd IEEE Computer Security Foundations Symp. (CSF '10)*. IEEE Computer Society, 186–199.

A. Sabelfeld and A. C. Myers. 2003. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19.

Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. 2018. Faceted Secure Multi Execution. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1617–1634. https://doi.org/10.1145/3243734.3243806

Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. 2012. Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems. In *International Conference on Functional Programming (ICFP)*. ACM SIGPLAN.

Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. 2017. Flexible Dynamic Information Flow Control in the Presence of Exceptions. *Journal of Functional Programming* 27 (2017).

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*.

Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. 2014. Protecting Users by Confining JavaScript with COWL. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

T. C. Tsai, A. Russo, and J. Hughes. 2007. A Library for Secure Multi-threaded Information Flow in Haskell. In *Proc. IEEE Computer Security Foundations Symposium (CSF '07)*.

Marco Vassena and Alejandro Russo. 2016. On Formalizing Information-Flow Control Libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/2993600.2993608

Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2017. MAC A Verified Static Information-Flow Control Library. *Journal of Logical and Algebraic Methods in Programming* (2017). https://doi.org/10.1016/j.jlamp.2017.12.003

Dennis Volpano and Geoffrey Smith. 1997. Eliminating Covert Flows with Minimum Typings. In *Proc. of the 10th IEEE workshop on Computer Security Foundations (CSFW '97)*. IEEE Computer Society.

Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A language for automatically enforcing privacy policies. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 137–148.

Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. 2009. Privacy-preserving Browser-side Scripting with BFlow. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM.

Stephan Arthur Zdancewic. 2002. *Programming languages for information security*. Ph.D. Dissertation. Cornell University.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in HiStar. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation*. USENIX.

Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. 2008. Securing Distributed Systems with Information Flow Control. (April 2008), 293–308 pages.

(Force)
$$\frac{e \Downarrow^{\theta} (t, \theta') \qquad \langle \Sigma, pc, t \rangle \Downarrow^{\theta'} \langle \Sigma', pc', v \rangle}{\langle \Sigma, pc, e \rangle \Downarrow^{\theta} \langle \Sigma', pc', v \rangle}$$

(a) Forcing semantics: $\langle \Sigma, pc, e \rangle \Downarrow^{\theta} \langle \Sigma', pc', v \rangle$.

(Thunk) $t \Downarrow^{\theta} (t, \theta)$

(Fun) $\lambda x.e \Downarrow^{\theta} (x.e, \theta)$

(Var) $x \Downarrow^{\theta} \theta(x)$

(App)
$$\frac{e_1 \Downarrow^{\theta} (x.e, \theta') \qquad e_2 \Downarrow^{\theta} v_2 \qquad e \Downarrow^{\theta'[x \mapsto v_2]} v}{e_1\ e_2 \Downarrow^{\theta} v}$$

(b) Pure semantics: $e \Downarrow^{\theta}$ v (selected rules).

(Return)
$$\frac{e \Downarrow^{\theta} v}{\langle \Sigma, pc, \mathbf{return}(e) \rangle \Downarrow^{\theta} \langle \Sigma, pc, v \rangle}$$

(Bind)
$$\frac{\langle \Sigma, pc, e_1 \rangle \Downarrow^{\theta} \langle \Sigma', pc', v_1 \rangle \qquad \langle \Sigma', pc', e_2 \rangle \Downarrow^{\theta[x \mapsto v_1]} \langle \Sigma'', pc'', v \rangle}{\langle \Sigma, pc, \mathbf{bind}(e_1, x.e_2) \rangle \Downarrow^{\theta} \langle \Sigma'', pc'', v \rangle}$$

(ToLabeled)
$$\frac{\langle \Sigma, pc, e \rangle \Downarrow^{\theta} \langle \Sigma', pc', v \rangle}{\langle \Sigma, pc, \mathbf{toLabeled}(e) \rangle \Downarrow^{\theta} \langle \Sigma', pc, \mathbf{Labeled}\ pc'\ v \rangle}$$

(Unlabel)
$$\frac{e \Downarrow^{\theta} \mathbf{Labeled}\ \ell\ v}{\langle \Sigma, pc, \mathbf{unlabel}(e) \rangle \Downarrow^{\theta} \langle \Sigma', pc \sqcup \ell, v \rangle}$$

(LabelOf)
$$\frac{e \Downarrow^{\theta} \mathbf{Labeled}\ \ell\ v}{\langle \Sigma, pc, \mathbf{labelOf}(e) \rangle \Downarrow^{\theta} \langle \Sigma, pc \sqcup \ell, \ell \rangle}$$

(GetLabel)
$$\langle \Sigma, pc, \mathbf{getLabel} \rangle \Downarrow^{\theta} \langle \Sigma, pc, pc \rangle$$

(Taint)
$$\frac{e \Downarrow^{\theta} \ell}{\langle \Sigma, pc, \mathbf{taint}(e) \rangle \Downarrow^{\theta} \langle \Sigma, pc \sqcup \ell, () \rangle}$$

(c) Thunk semantics: $\langle \Sigma, pc, t \rangle \Downarrow^{\theta} \langle \Sigma', pc', v \rangle$.

Fig. 7. Semantics of λ^{dCG}.

$$\textsc{New}\ \frac{e \Downarrow^{\theta} \textbf{Labeled}\ \ell\ v \qquad pc \sqsubseteq \ell \qquad n = |\Sigma(\ell)|}{\langle \Sigma, pc, \textbf{new}(e) \rangle \Downarrow^{\theta} \langle \Sigma[\ell \mapsto \Sigma(\ell)[n \mapsto v]], pc, n_{\ell} \rangle}$$

$$\textsc{Read}\ \frac{e \Downarrow^{\theta} n_{\ell} \qquad \Sigma(\ell)[n] = v}{\langle \Sigma, pc, !e \rangle \Downarrow^{\theta} \langle \Sigma, pc \sqcup \ell, v \rangle}$$

$$\textsc{Write}\ \frac{e_1 \Downarrow^{\theta} n_{\ell_1} \qquad e_2 \Downarrow^{\theta} \textbf{Labeled}\ \ell_2\ v \qquad \ell_2 \sqsubseteq \ell_1 \qquad pc \sqsubseteq \ell_1}{\langle \Sigma, pc, e_1 := e_2 \rangle \Downarrow^{\theta} \langle \Sigma[\ell_1 \mapsto \Sigma(\ell_1)[n \mapsto v]], pc, () \rangle}$$

$$\textsc{LabelOfRef}\ \frac{e \Downarrow^{\theta} n_{\ell}}{\langle \Sigma, pc, \textbf{labelOfRef}(e) \rangle \Downarrow^{\theta} \langle \Sigma, pc \sqcup \ell, \ell \rangle}$$

Fig. 8. λ^{dCG} semantics for operations on references.

$$(\textsc{Labeled}_L)\ \frac{\ell \sqsubseteq L \qquad v_1 \approx_L v_2}{\textbf{Labeled}\ \ell\ v_1 \approx_L \textbf{Labeled}\ \ell\ v_2} \qquad (\textsc{Labeled}_H)\ \frac{\ell_1 \not\sqsubseteq L \qquad \ell_2 \not\sqsubseteq L}{\textbf{Labeled}\ \ell_1\ v_1 \approx_L \textbf{Labeled}\ \ell_2\ v_2}$$

$$(\textsc{Closure})\ \frac{e_1 \equiv_\alpha e_2 \qquad \theta_1 \approx_L \theta_2}{(e_1, \theta_1) \approx_L (e_2, \theta_2)} \qquad (\textsc{Thunk})\ \frac{t_1 \equiv_\alpha t_2 \qquad \theta_1 \approx_L \theta_2}{(t_1, \theta_1) \approx_L (t_2, \theta_2)} \qquad (\textsc{Ref}_L)\ \frac{\ell \sqsubseteq L}{n^{\ell} \approx_L n^{\ell}} \qquad (\textsc{Ref}_H)\ \frac{\ell_1 \not\sqsubseteq L \qquad \ell_2 \not\sqsubseteq L}{{n_1}^{\ell_1} \approx_L {n_2}^{\ell_2}}$$

$$(\textsc{Pc}_H)\ \frac{\Sigma_1 \approx_L \Sigma_2 \qquad pc_1 \not\sqsubseteq L \qquad pc_2 \not\sqsubseteq L}{\langle \Sigma_1, pc_1, v_1 \rangle \approx_L \langle \Sigma_2, pc_2, v_2 \rangle} \qquad (\textsc{Pc}_L)\ \frac{\Sigma_1 \approx_L \Sigma_2 \qquad pc \sqsubseteq L \qquad v_1 \approx_L v_2}{\langle \Sigma_1, pc, v_1 \rangle \approx_L \langle \Sigma_2, pc, v_2 \rangle}$$

Fig. 9. L-equivalence for λ^{dCG} values (selected rules) and configurations.

$$\begin{aligned}
\langle\!\langle \textbf{unit} \rangle\!\rangle &= \textbf{Labeled unit}\\
\langle\!\langle \mathscr{L} \rangle\!\rangle &= \textbf{Labeled}\ \mathscr{L}\\
\langle\!\langle \tau_1 \times \tau_2 \rangle\!\rangle &= \textbf{Labeled}\ (\langle\!\langle \tau_1 \rangle\!\rangle \times \langle\!\langle \tau_2 \rangle\!\rangle)\\
\langle\!\langle \tau_1 + \tau_2 \rangle\!\rangle &= \textbf{Labeled}\ (\langle\!\langle \tau_1 \rangle\!\rangle + \langle\!\langle \tau_2 \rangle\!\rangle)\\
\langle\!\langle \tau_1 \to \tau_2 \rangle\!\rangle &= \textbf{Labeled}\ (\langle\!\langle \tau_1 \rangle\!\rangle \to \textbf{LIO} \langle\!\langle \tau_2 \rangle\!\rangle)\\
\langle\!\langle \textbf{Ref}\ \tau \rangle\!\rangle &= \textbf{Labeled}\ (\textbf{Ref} \langle\!\langle \tau \rangle\!\rangle)
\end{aligned}$$

Fig. 10. Type translation from λ^{dFG} to λ^{dCG}.

$$\begin{aligned}
\langle\!\langle r^{\ell} \rangle\!\rangle &= \textbf{Labeled}\ \ell \langle\!\langle r \rangle\!\rangle\\
\langle\!\langle () \rangle\!\rangle &= ()\\
\langle\!\langle \ell \rangle\!\rangle &= \ell\\
\langle\!\langle (v_1, v_2) \rangle\!\rangle &= (\langle\!\langle v_1 \rangle\!\rangle, \langle\!\langle v_2 \rangle\!\rangle)\\
\langle\!\langle \textbf{inl}(v) \rangle\!\rangle &= \textbf{inl}(\langle\!\langle v \rangle\!\rangle)\\
\langle\!\langle \textbf{inr}(v) \rangle\!\rangle &= \textbf{inr}(\langle\!\langle v \rangle\!\rangle)\\
\langle\!\langle (x.e, \theta) \rangle\!\rangle &= (x.\langle\!\langle e \rangle\!\rangle, \langle\!\langle \theta \rangle\!\rangle)\\
\langle\!\langle n_{\ell} \rangle\!\rangle &= n_{\ell}
\end{aligned}$$

Fig. 11. Value translation from λ^{dFG} to λ^{dCG}.

$$
\begin{array}{ll}
\langle\!\langle () \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{return}(())) & \langle\!\langle \mathbf{case}(e, x.e_1, x.e_2) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} \\
\langle\!\langle \ell \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{return}(\ell)) & \quad lv \leftarrow \langle\!\langle e \rangle\!\rangle \\
\langle\!\langle (\lambda x.e) \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{return}(\lambda x.\langle\!\langle e \rangle\!\rangle)) & \quad v \leftarrow \mathbf{unlabel}(lv) \\
\langle\!\langle \mathbf{inl}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} & \quad lv' \leftarrow \mathbf{case}(v, x.\langle\!\langle e_1 \rangle\!\rangle, x.\langle\!\langle e_2 \rangle\!\rangle) \\
\quad le \leftarrow \langle\!\langle e \rangle\!\rangle & \quad \mathbf{unlabel}(lv')) \\
\quad \mathbf{return}(\mathbf{inl}(lv))) & \langle\!\langle \mathbf{fst}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} \\
\langle\!\langle \mathbf{inr}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} & \quad lv \leftarrow \langle\!\langle e \rangle\!\rangle \\
\quad le \leftarrow \langle\!\langle e \rangle\!\rangle & \quad v \leftarrow \mathbf{unlabel}(lv) \\
\quad \mathbf{return}(\mathbf{inr}(lv))) & \quad \mathbf{unlabel}(\mathbf{fst}(v))) \\
\langle\!\langle (e_1, e_2) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} & \langle\!\langle \mathbf{snd}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} \\
\quad lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle & \quad lv \leftarrow \langle\!\langle e \rangle\!\rangle \\
\quad lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle & \quad v \leftarrow \mathbf{unlabel}(lv) \\
\quad \mathbf{return}(lv_1, lv_2)) & \quad \mathbf{unlabel}(\mathbf{snd}(v))) \\
\langle\!\langle x \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{unlabel}(x)) & \langle\!\langle \mathbf{taint}(e_1, e_2) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} \\
\langle\!\langle e_1\ e_2 \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} & \quad lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle \\
\quad lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle & \quad v_1 \leftarrow \mathbf{unlabel}(lv_1) \\
\quad lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle & \quad \mathbf{taint}(v_1) \\
\quad v_1 \leftarrow \mathbf{unlabel}(lv_1) & \quad lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle \\
\quad lv \leftarrow v_1\ lv_2 & \quad \mathbf{unlabel}(lv_2)) \\
\quad \mathbf{unlabel}(lv)) & \langle\!\langle \mathbf{labelOf}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} \\
 & \quad lv \leftarrow \langle\!\langle e \rangle\!\rangle \\
 & \quad \mathbf{labelOf}(lv)) \\
 & \langle\!\langle \mathbf{getLabel} \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{getLabel})
\end{array}
$$

Fig. 12. Expression translation from λ^{dFG} to λ^{dCG}.

$$
\begin{array}{lll}
\langle\!\langle \mathbf{new}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} & \langle\!\langle e_1 := e_2 \rangle\!\rangle = & \langle\!\langle \mathbf{labelOfRef}(e) \rangle\!\rangle = \\
\quad lv \leftarrow \langle\!\langle e \rangle\!\rangle & \quad \mathbf{toLabeled}(\ \mathbf{do} & \quad \mathbf{toLabeled}(\ \mathbf{do} \\
\quad \mathbf{new}(lv)) & \quad\quad lr \leftarrow \langle\!\langle e_1 \rangle\!\rangle & \quad\quad lr \leftarrow \langle\!\langle e \rangle\!\rangle \\
\langle\!\langle\, !\, e \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do} & \quad\quad lv \leftarrow \langle\!\langle e_2 \rangle\!\rangle & \quad\quad r \leftarrow \mathbf{unlabel}(lv) \\
\quad lr \leftarrow \langle\!\langle e \rangle\!\rangle & \quad\quad r \leftarrow \mathbf{unlabel}(lr) & \quad\quad \mathbf{labelOfRef}(r)) \\
\quad r \leftarrow \mathbf{unlabel}(lv) & \quad\quad r := lv) & \\
\quad !\, r) & \quad \mathbf{toLabeled}(\mathbf{return}()) &
\end{array}
$$

Fig. 13. λ^{dFG} to λ^{dCG} translation of memory operations.

$$
\begin{aligned}
&[\![\mathscr{L}]\!] = \mathscr{L}\\
&[\![\mathbf{unit}]\!] = \mathbf{unit}\\
&[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!]\\
&[\![\tau_1 + \tau_2]\!] = [\![\tau_1]\!] + [\![\tau_2]\!]\\
&[\![\tau_1 \times \tau_2]\!] = [\![\tau_1]\!] \times [\![\tau_2]\!]\\
&[\![\mathbf{Ref}\ \tau]\!] = \mathbf{Ref}[\![\tau]\!]\\
&[\![\mathbf{Labeled}\ \tau]\!] = \mathscr{L} \times [\![\tau]\!]\\
&[\![\mathbf{LIO}\ \tau]\!] = \mathbf{unit} \to [\![\tau]\!]
\end{aligned}
$$

Fig. 14. Type translation from λ^{dCG} to λ^{dFG}.

$$
\begin{aligned}
&[\![()]\!]^{pc} = ()^{pc}\\
&[\![\ell]\!]^{pc} = \ell^{pc}\\
&[\![\mathbf{inl}(v)]\!]^{pc} = \mathbf{inl}([\![v]\!]^{pc})^{pc}\\
&[\![\mathbf{inr}(v)]\!]^{pc} = \mathbf{inr}([\![v]\!]^{pc})^{pc}\\
&[\![(v_1, v_2)]\!]^{pc} = ([\![v_1]\!]^{pc}, [\![v_2]\!]^{pc})^{pc}\\
&[\![(x.e, \theta)]\!]^{pc} = (x.[\![e]\!], [\![\theta]\!]^{pc})^{pc}\\
&[\![(t, \theta)]\!]^{pc} = (\_.[\![t]\!], [\![\theta]\!]^{pc})^{pc}\\
&[\![\mathbf{Labeled}\ \ell\ v]\!]^{pc} = (\ell^{\ell}, [\![v]\!]^{\ell})^{pc}\\
&[\![n_\ell]\!]^{pc} = (n_\ell)^{pc}
\end{aligned}
$$

Fig. 15. Value translation from λ^{dCG} to λ^{dFG}.

$$
\begin{aligned}
&[\![()]\!] = ()\\
&[\![\ell]\!] = \ell\\
&[\![x]\!] = x\\
&[\![\lambda x.e]\!] = \lambda x.[\![e]\!]\\
&[\![e_1\ e_2]\!] = [\![e_1]\!][\![e_2]\!]\\
&[\![(e_1, e_2)]\!] = ([\![e_1]\!], [\![e_2]\!])\\
&[\![\mathbf{fst}(e)]\!] = \mathbf{fst}([\![e]\!])\\
&[\![\mathbf{snd}(e)]\!] = \mathbf{snd}([\![e]\!])\\
&[\![\mathbf{inl}(e)]\!] = \mathbf{inl}([\![e]\!])\\
&[\![\mathbf{inr}(e)]\!] = \mathbf{inr}([\![e]\!])\\
&[\![\mathbf{case}\ (e, x.e_1, x.e_2)]\!]\\
&\quad = \mathbf{case}\ ([\![e]\!], x.[\![e_1]\!], x.[\![e_2]\!])\\
&[\![t]\!] = \lambda\_.[\![t]\!]
\end{aligned}
$$

Fig. 16. Expr. translation from λ^{dCG} to λ^{dFG}.

$$
\begin{aligned}
&[\![\mathbf{return}(e)]\!] = [\![e]\!]\\
&[\![\mathbf{bind}(e_1, x.e_2)]\!] =\\
&\quad \mathbf{let}\ x = [\![e_1]\!]()\ \mathbf{in}\\
&\qquad \mathbf{taint}(\mathbf{labelOf}(x), [\![e_2]\!]())\\
&[\![\mathbf{unlabel}(e)]\!] =\\
&\quad \mathbf{let}\ x = [\![e]\!]\mathbf{in}\\
&\qquad \mathbf{taint}(\mathbf{fst}(x), \mathbf{snd}(x))\\
&[\![\mathbf{toLabeled}(e)]\!] =\\
&\quad \mathbf{let}\ x = [\![e]\!]()\ \mathbf{in}\\
&\qquad (\mathbf{labelOf}(x), x)\\
&[\![\mathbf{labelOf}(e)]\!] = \mathbf{fst}([\![e]\!])\\
&[\![\mathbf{getLabel}]\!] = \mathbf{getLabel}\\
&[\![\mathbf{taint}(e)]\!] = \mathbf{taint}([\![e]\!], ())
\end{aligned}
$$

Fig. 17. Thunk translation λ^{dCG} to λ^{dFG}.

$$
\begin{aligned}
&[\![\mathbf{new}(e)]\!] =\\
&\quad \mathbf{let}\ x = [\![e]\!]\mathbf{in}\\
&\qquad \mathbf{new}(\mathbf{taint}(\mathbf{fst}(x), \mathbf{snd}(x)))
\end{aligned}
\qquad
\begin{aligned}
&[\![e_1 := e_2]\!] = [\![e_1]\!] := \mathbf{snd}([\![e_2]\!])\\
&[\![\,!\,e]\!] = ![\![e]\!]\\
&[\![\mathbf{labelOfRef}(e)]\!] = \mathbf{labelOfRef}([\![e]\!])
\end{aligned}
$$

Fig. 18. λ^{dCG} to λ^{dFG} translation of memory operations.

$$\text{(Value)}\ \frac{\ell_1 \sqsubseteq pc \qquad r_1 \downarrow\approx_{pc} v_2}{{r_1}^{\ell_1} \downarrow\approx_{pc} v_2} \qquad \text{(Unit)}\ () \downarrow\approx_{pc} () \qquad \text{(Label)}\ \ell \downarrow\approx_{pc} \ell \qquad \text{(Ref)}\ n_\ell \downarrow\approx_{pc} n_\ell$$

$$\text{(Inl)}\ \frac{v_1 \downarrow\approx_{pc} v_1'}{\mathbf{inl}(v_1) \downarrow\approx_{pc} \mathbf{inl}(v_1')} \qquad \text{(Inr)}\ \frac{v_2 \downarrow\approx_{pc} v_2'}{\mathbf{inr}(v_2) \downarrow\approx_{pc} \mathbf{inr}(v_2')} \qquad \text{(Pair)}\ \frac{v_1 \downarrow\approx_{pc} v_1' \qquad v_2 \downarrow\approx_{pc} v_2'}{(v_1, v_2) \downarrow\approx_{pc} (v_1', v_2')}$$

$$\text{(Fun)}\ \frac{\theta_1 \downarrow\approx_{pc} \theta_2}{(x.[\![e]\!], \theta_1) \downarrow\approx_{pc} (x.e, \theta_2)} \qquad \text{(Thunk)}\ \frac{\theta_1 \downarrow\approx_{pc} \theta_2}{(\_.[\![t]\!], \theta_1) \downarrow\approx_{pc} (t, \theta_2)} \qquad \text{(Labeled)}\ \frac{v_1 \downarrow\approx_{\ell} v_2}{(\ell^{\ell}, v_1) \downarrow\approx_{pc} (\mathbf{Labeled}\ \ell\ v_2)}$$

Fig. 19. Cross-language value equivalence modulo label annotations.