Foundations and Trends® in Programming Languages

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

## A Tutorial on Dynamic Information Flow

**Suggested Citation:** Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani and Deian Stefan (2023), "From Fine- to Coarse-Grained Dynamic Information Flow Control and Back", Foundations and Trends® in Programming Languages: Vol. 8, No. 1, pp 1–117. DOI: 10.1561/2500000046.

**Marco Vassena**
Utrecht University
m.vassena@uu.nl

**Alejandro Russo**
Chalmers University of Technology
russo@chalmers.se

**Deepak Garg**
Max Planck Institute for Software Systems
dg@mpi-sws.org

**Vineet Rajani**
University of Kent
v.rajani@kent.ac.uk

**Deian Stefan**
University of California, San Diego
deian@cs.ucsd.edu



now
the essence of knowledge
Boston — Delft

# Contents

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

Marco Vassena[1], Alejandro Russo[2], Deepak Garg[3], Vineet Rajani[4] and Deian Stefan[5]

[1] *Utrecht University, The Netherlands; m.vassena@uu.nl*
[2] *Chalmers University of Technology, Sweden; russo@chalmers.se*
[3] *Max Planck Institute for Software Systems, Germany; dg@mpi-sws.org*
[4] *University of Kent, UK; v.rajani@kent.ac.uk*
[5] *University of California, San Diego, USA; deian@cs.ucsd.edu*

ABSTRACT

This tutorial provides a complete and homogeneous account of the latest advances in fine- and coarse-grained dynamic information-flow control (IFC) security. Since the 1970s, the programming language and the operating system communities proposed different IFC approaches. IFC operating systems track information flows in a coarse-grained fashion, at the granularity of a process. In contrast, traditional language-based approaches to IFC are fine-grained: they track information flows at the granularity of program variables. For decades, researchers believed coarse-grained IFC to be strictly less permissive than fine-grained IFC—coarse grained IFC systems seem inherently less precise because they track less information——and so granularity appeared to be a fundamental feature of IFC systems.

Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani and Deian Stefan (2023), "From Fine- to Coarse-Grained Dynamic Information Flow Control and Back", Foundations and Trends® in Programming Languages: Vol. 8, No. 1, pp 1–117.
DOI: 10.1561/2500000046.

We show that the granularity of the tracking system does *not* fundamentally restrict how precise or permissive dynamic IFC systems can be. 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.

These translations 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.

# 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—the *granularity* of the tracking system alone has important implications for the usage of IFC technology. 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; Zeldovich *et al.*, 2006; Krohn *et al.*, 2007) 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; Zdancewic, 2002; Hedin *et al.*, 2014; Hritcu *et al.*, 2013; Yang *et al.*, 2012) track the flow of information in a more *fine-grained* fashion, e.g., at the granularity of program variables and references.

Dynamic coarse-grained IFC systems in the style of LIO (Stefan *et al.*, 2017; Stefan *et al.*, 2011; Stefan *et al.*, 2012; Heule *et al.*, 2015; Buiras *et al.*, 2015; 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 a single label-annotation 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, coarse-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 tutorial 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) and Rajani and Garg (2018), who prove similar results for *static* fine-grained and coarse-grained IFC systems. Specifi-

cally, 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 the Agda proof assistant (Norell, 2009; Bove *et al.*, 2009), 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), i.e., they can be expressed as a pure source program rewriting.

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.* (2017) and Stefan *et al.* (2012), provide **toLabeled**$(\cdot)$ 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.

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 monograph are:

- A pair of semantics-preserving translations between traditional dynamic fine-grained and coarse-grained IFC systems equipped with label introspection and flow-insensitive references (Theorems 5 and 7).
- Two different proofs of *termination-insensitive* non-interference (TINI) for each calculus: one is derived directly in the usual way (Theorems 1 and 3), while the other is recovered via our verified translation (Theorems 6 and 8).
- Mechanized Agda proofs of our results (~4,000 LOC).

This monograph is based on our conference paper (Vassena *et al.*, 2019) and extended with:

- A tutorial-style introduction to fine- and coarse-grained dynamic IFC, which (i) illustrates their specific features and (apparent) differences through examples, and (ii) supplements our proof artifacts with general explanations of the proof techniques used.
- *Flow-sensitive* references, a key feature for boosting the permissiveness of dynamic IFC systems (Austin and Flanagan, 2009). We extend both fine- and coarse-grained language with flow-sensitive references (Sections 2.3 and 3.3), adapt their security proofs (Theorems 2 and 4), and the verified translations to each other.
- A discussion and analysis of our extended proof artifact (~6,900 LOC)[1]. Our analysis finds that the security proofs for fine-grained languages are between 43% and 74% longer than for coarse-grained languages. These empirical results suggests that it is indeed easier to reason about coarse-grained IFC languages than fine-grained languages.

[1]The extended artifact is available at https://hub.docker.com/r/marcovassena/granularity-ftpl and supersedes the artifact archived with the conference paper.

This tutorial is organized as follows. Our dynamic fine- and coarse-grained IFC calculi are introduced in Sections 2 and 3, and then extended with flow-sensitive references in Sections 2.3 and 3.3, respectively. We prove the soundness guarantees (i.e., termination-insensitive non-interference) of the original languages (Sections 2.2 and 3.2), and of the extended languages (Sections 2.3.3 and 3.3.3). In Section 4, we discuss our mechanized proof artifacts and compare the security proofs of the two calculi, before and after the extension. In Section 5, we present the fine- to coarse-grained translation and a proof of non-interference for the fine-grained calculus recovered from non-interference of the other calculus through our verified translation. Section 6 presents similar results in the other direction. Related work is described in Section 7 and Section 8 concludes the tutorial.

# 2

# Fine-Grained IFC Calculus

In order to compare in a rigorous way fine- and coarse-grained dynamic IFC techniques, we formally define the operational semantics of two $\lambda$-calculi that respectively perform fine- and coarse-grained IFC dynamically. Figure 2.1 shows the syntax of the dynamic fine-grained IFC calculus $\lambda^{dFG}$, which is inspired by Austin and Flanagan (2009) and extended with a standard (security unaware) type system $\Gamma \vdash e : \tau$ (omitted), sum and product data types and security labels $\ell \in \mathscr{L}$ that form a lattice $(\mathscr{L}, \sqsubseteq)$.[1] In order to capture flows of information precisely at run-time, the $\lambda^{dFG}$-calculus features *intrinsically labeled* values, written $r^{\ell}$, meaning that raw value $r$ has security level $\ell$. 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 $(\mathbf{true}^{H}, \mathbf{false}^{L})$.[2] 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 $\theta$. The $\lambda^{dFG}$-calculus features

[1] 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., $H \not\sqsubseteq L$.

[2] 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 $\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 = \mathbf{case}\ e\ \_.e_1\ \_.e_2$.

$$
\begin{array}{rl}
\text{Types:} & \tau ::= \mathbf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mathscr{L} \mid \mathbf{Ref}\ \tau \\
\text{Labels:} & \ell, pc \in \mathscr{L} \\
\text{Addresses:} & n \in \mathbb{N} \\
\text{Environments:} & \theta \in \mathit{Var} \rightharpoonup \mathit{Value} \\
\text{Raw Values:} & r ::= () \mid (x.e, \theta) \mid \mathbf{inl}(v) \mid \mathbf{inr}(v) \mid (v_1, v_2) \mid \ell \mid n_\ell \\
\text{Values:} & v ::= r^\ell \\
\text{Expression:} & e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid () \mid \ell \mid (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \\
& \mid \mathbf{inl}(e) \mid \mathbf{inr}(e) \mid \mathbf{case}(e, x.e_1, x.e_2) \\
& \mid \mathbf{getLabel} \mid \mathbf{labelOf}(e) \mid e_1 \sqsubseteq^?\ e_2 \mid \mathbf{taint}(e_1, e_2) \\
& \mid \mathbf{new}(e) \mid !\, e \mid e_1 := e_2 \mid \mathbf{labelOfRef}(e) \\
\text{Type System:} & \Gamma \vdash e : \tau \\
\text{Configurations:} & c ::= \langle \Sigma, e \rangle \\
\text{Stores:} & \Sigma \in (\ell : \mathit{Label}) \to \mathit{Memory}\ \ell \\
\text{Memory } \ell\text{:} & M ::= [\,] \mid r : M
\end{array}
$$

**Figure 2.1:** Syntax of $\lambda^{dFG}$.

a labeled partitioned store, i.e., $\Sigma \in (\ell : \mathscr{L}) \to \mathit{Memory}\ \ell$, where $\mathit{Memory}\ \ell$ is the memory that contains values at security level $\ell$. Each reference carries an additional label annotation that records the label of the memory it refers to—reference $n_\ell$ points to the $n$-th cell of the $\ell$-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 IFC runtime monitor 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 $\lambda^{dFG}$ 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.[3] In Figure 2.2, the relation $\langle \Sigma, e \rangle \Downarrow^{\theta}_{pc} \langle \Sigma', v \rangle$ denotes the evaluation of program $e$ with initial store $\Sigma$ that terminates with labeled value $v$ and final store $\Sigma'$. The environment $\theta$ 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, 2006)— 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 (i) propagate and assign labels to values computed by a program, and (ii) 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 $\theta$.

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., **case** $e\ x.e_1\ x.e_2$. Rules [Case$_1$] and [Case$_2$] evaluate the scrutinee $e$ to a value (either $\mathbf{inl}(v)^{\ell}$ or $\mathbf{inr}(v)^{\ell}$), add the value to the environment, i.e., $\theta[x \mapsto v]$, and then evaluate 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

[3] In this work, we ignore leaks that exploit program termination and prove *termination insensitive* non-interference for $\lambda^{dFG}$ (Theorem 1).

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

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

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

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

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

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

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

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

**Figure 2.2:** Big-step semantics for $\lambda^{dFG}$ (part I).

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

$$(\sqsubseteq^{?}\text{-T})\quad \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \ell_1{}^{\ell'_1}\rangle \qquad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', \ell_2{}^{\ell'_2}\rangle \qquad \ell_1 \sqsubseteq \ell_2}{\langle \Sigma, e_1 \sqsubseteq^{?} e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', \mathbf{inl}((\,)^{pc})^{\ell'_1 \sqcup \ell'_2}\rangle}$$

$$(\sqsubseteq^{?}\text{-F})\quad \frac{\langle \Sigma, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \ell_1{}^{\ell'_1}\rangle \qquad \langle \Sigma', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', \ell_2{}^{\ell'_2}\rangle \qquad \ell_1 \not\sqsubseteq \ell_2}{\langle \Sigma, e_1 \sqsubseteq^{?} e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', \mathbf{inr}((\,)^{pc})^{\ell'_1 \sqcup \ell'_2}\rangle}$$

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

**Figure 2.3:** Big-step semantics for $\lambda^{dFG}$ (part II).

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 $\ell$, 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 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., $(\lambda x.())\ y \ \Downarrow^{[y \mapsto \mathbf{true}^{H}]}_{L}\ ()^{L}$. 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 $\ell'$—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 $x \ \Downarrow^{[x \mapsto ()^{L}]}_{H}\ ()^{H}$.

### 2.1.1 Label Introspection

The $\lambda^{dFG}$-calculus features primitives for label introspection, namely **getLabel**, **labelOf**$(\cdot)$ and $\sqsubseteq^{?}$—see Figure 2.3. 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 $\lambda^{dFG}$ 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 $\lambda^{dFG}$, 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., $\ell^{\ell}$ and $pc^{pc}$ in rules [GetLabel] and [LabelOf] in Figure 2.3. 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 5 uses **taint**$(\cdot)$ to loosen the precision of $\lambda^{dFG}$ in a controlled way and match the *coarse* approximation of our coarse-grained IFC calculus ($\lambda^{dCG}$) by upgrading the labels of intermediate values systematically. In rule [Taint], the constraint $\ell' \sqsubseteq \ell$ ensures that the label of the nested context $\ell$ 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 of any $\lambda^{dFG}$ program is always 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 $pc \sqsubseteq \ell$.

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

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

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

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

**Figure 2.4:** Big-step semantics for $\lambda^{dFG}$ (references).

*Proof.* By induction on the given evaluation derivation. □

### 2.1.2 References

We now extend the semantics presented earlier with primitives that inspect, access and modify the labeled store via labeled references. See Figure 2.4. Rule [New] creates a reference $n_\ell$, labeled with the security level of the initial content, i.e., label $\ell$, in the $\ell$-labeled memory $\Sigma(\ell)$ and updates the memory store accordingly.[4] 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 $n_\ell$ at security level $\ell'$, rule [Read] retrieves the corresponding raw value from the $n$-th cell of the $\ell$-labeled memory, i.e., $\Sigma'(\ell)[n] = r$ and upgrades its label to $\ell \sqcup \ell'$ since the decision to read from that

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

particular reference depends on information at security level $\ell'$. 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., $r^{\ell_2}$, 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*),[5] 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 via an *explicit flow*). 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 level of the reference, i.e., $\ell^{\ell \sqcup \ell'}$ to avoid leaks. Intuitively, the label of the reference, i.e., $\ell$, depends also on data at security level $\ell'$ as seen in the premise.

**Other Extensions.** We consider $\lambda^{dFG}$ 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 5 and 6. 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., Russo *et al.*, 2009; Stefan *et al.*, 2011; Efstathopoulos *et al.*, 2005; Krohn *et al.*, 2007) enforcements are likely to assign a single *flow-insensitive* (i.e., immutable) 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., Pottier and Simonet, 2003; Myers *et al.*, 2006; Stefan *et al.*, 2011; Vassena and Russo, 2016), in a manner similar to what we have just shown for references in $\lambda^{dFG}$.

Importantly, fine- and coarse-grained IFC are equally expressive also when extended with *flow-sensitive* (i.e., mutable) labels. Unlike

[5] 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** ().

flow-insensitive labels, these labels can *change* during program execution to reflect the current sensitivity of the content of various resources (e.g., references, files etc.). In order to show that fine- and coarse-grained IFC equally support flow-sensitive features, we follow the same approach described above. In particular, we add *flow-sensitive* references (Austin and Flanagan, 2009) to $\lambda^{dFG}$ (Section 2.3) and $\lambda^{dCG}$ (Section 3.3) and then complete our pair of verified semantics- and security-preserving translations from one language to the other (Sections 5 and 6).

## 2.2 Security

We now prove that $\lambda^{dFG}$ 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 $\lambda^{dFG}$ 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.

### 2.2.1 $L$-Equivalence

$L$-equivalence for values and raw-values is defined formally by mutual induction in Figure 2.5. 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 (i.e., rules [Unit] and [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 through rules [Inl] and [Inr]. Closures are related by rule [Closure], if they contain the *same* function (up to $\alpha$-renaming)[6] 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 $\forall x.\theta_1(x) \approx_L \theta_2(x)$.

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

$$\frac{}{() \approx_L ()}\ (\textsc{Unit}) \qquad \frac{}{\ell \approx_L \ell}\ (\textsc{Label})$$

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

$$\frac{v_1 \approx_L v_1' \qquad v_2 \approx_L v_2'}{(v_1, v_2) \approx_L (v_1', v_2')}\ (\textsc{Pair}) \qquad \frac{\ell \sqsubseteq L}{n_\ell \approx_L n_\ell}\ (\textsc{Ref}_L) \qquad \frac{\ell_1 \not\sqsubseteq L \qquad \ell_2 \not\sqsubseteq L}{n_{1\ell_1} \approx_L n_{2\ell_2}}\ (\textsc{Ref}_H)$$

**Figure 2.5:** $L$-equivalence for $\lambda^{dFG}$ values and raw values.

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 $\ell$-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 through rule [$\textsc{Ref}_H$], but requires the same label and address for public references in rule [$\textsc{Ref}_L$]. 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 $v_1 \approx_L v_2$.

The $L$-equivalence relation defined above is *reflexive*, *symmetric*, and *transitive*.

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

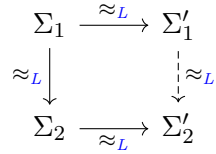


**Figure 2.6:** Square commutative diagram for stores (Lemma 2.1).

**Property 2.** Let $x$, $y$, $z$ range over labeled values, raw values, environments, labeled memories, stores, and configurations:

1. **Reflexivity.** For all $x$, $x \approx_L x$.
2. **Symmetricity.** For all $x$ and $y$, if $x \approx_L y$, then $y \approx_L x$.
3. **Transitivity.** For all $x$, $y$, $z$, if $x \approx_L y$ and $y \approx_L z$, then $x \approx_L z$.

These properties simplify the security analysis of $\lambda^{dFG}$: they let us reason about $L$-equivalent terms using *commutative diagrams*. For example, consider the *Square Commutative Diagram for Stores* outlined in Figure 2.6. In the diagram, the arrows connect $L$-equivalent stores (e.g., the arrow from $\Sigma_1$ to $\Sigma'_1$ indicates that $\Sigma_1 \approx_L \Sigma'_1$). The diagram provides a visual representation of Lemma 2.1: solid arrows represent the assumptions of the lemma and the dashed arrow represents the conclusion. To prove the lemma, we have to show that the diagram *commutes*, i.e., we need to construct a path from $\Sigma'_1$ to $\Sigma'_2$ using the solid arrows. Thanks to Property 2, we can derive additional arrows to construct this path. For example, we can reverse arrows using *symmetricity* (e.g., $\Sigma'_1 \approx_L \Sigma_1$ from $\Sigma_1 \approx_L \Sigma'_1$) and *transitivity* let us compose consecutive arrows (e.g., $\Sigma_1 \approx_L \Sigma'_2$ from $\Sigma_1 \approx_L \Sigma_2$ and $\Sigma_2 \approx_L \Sigma'_2$).

**Lemma 2.1** (Square Commutative Diagram for Stores)**.** If $\Sigma_1 \approx_L \Sigma'_1$, $\Sigma_1 \approx_L \Sigma_2$, $\Sigma_2 \approx_L \Sigma'_2$, then $\Sigma'_1 \approx_L \Sigma'_2$.

*Proof.* We show that the square diagram commutes using *symmetricity* (Property 2.2) and *transitivity* (Property 2.3) to draw the red arrows in Figure 2.7.

□

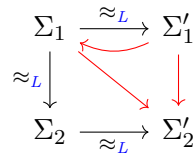


**Figure 2.7:** Proof of Lemma 2.1. The red arrows represent L-equivalent relations derived via symmetricity and transitivity.

### 2.2.2 Termination-Insensitive Non-Interference

The security monitor of $\lambda^{dFG}$ enforces termination-insensitive non-interference. Intuitively, this property guarantees that *terminating* programs do not leak secret data into public values and memories of the store. More formally, a program satisfies non-interference if, given indistinguishable inputs (initial stores and environments), then it produces outputs (final stores and values) that are also indistinguishable to the attacker. We prove this result through two key lemmas: *store confinement* and *$L$-equivalence preservation*. In the following, we give a high-level overview of these lemmas and their proof, focusing on the general proof technique and illustrative cases. We refer to our mechanized proof scripts for complete proofs.

**Store Confinement.** At a high-level, store confinement ensures that programs cannot leak secret data *implicitly* through observable side-effects in the labeled store. Intuitively, the side-effects of programs running in secret contexts must be *confined* to secret memories in the labeled store to enforce security—programs that do otherwise may leak and are therefore conservatively aborted by the security monitor. This lemma holds for $\lambda^{dFG}$ precisely because the constraints in rules [New] and [Write] only allow programs to write memories labeled above the program counter label. In particular, these constraints prevent programs running in secret contexts from writing public memories, which remain unchanged and thus indistinguishable to the attacker.

**Lemma 2.2** (Store Confinement)**.** For all configurations $c = \langle \Sigma, e \rangle$, $c' = \langle \Sigma', v \rangle$, program counter labels $pc \not\sqsubseteq L$, if $c \Downarrow^{\theta}_{pc} c'$, then $\Sigma \approx_L \Sigma'$.

*Proof.* The proof is by induction on the big-step reduction, using *reflexivity* (Property 2.1) in the base cases and *transitivity* (Property 2.3) in the inductive cases. In the inductive cases (e.g., [CASE$_1$]), we observe that the program counter label of the nested computations is always *at least as sensitive* as the initial program counter $pc$, and therefore above the attacker's label $L$, i.e., if $pc \not\sqsubseteq L$, then $pc \sqcup \ell \not\sqsubseteq L$ for any label $\ell$. The interesting cases are those that change the store, i.e., cases [NEW] and [WRITE], where we use Property 1 to show that these rules can modify only secret memories. For example, in case [NEW], the program creates a reference in a secret context ($pc \not\sqsubseteq L$). First, the program computes a value labeled $\ell$ above the attacker's label $L$, i.e., $\ell \not\sqsubseteq L$ by Property 1, and then allocates it in the corresponding secret memory also labeled $\ell$. Since $\ell \not\sqsubseteq L$, the original and the extended memory are indistinguishable by the attacker and so are the initial and final stores. □

***L*-equivalence Preservation.** Termination-insensitive non-interference ensures that programs that receive $L$-equivalent inputs produce outputs that are also $L$-equivalent, i.e., terminating programs *preserve* $L$-equivalence. Importantly, programs must preserve $L$-equivalence regardless of the sensitivity of the context in which they are executed. Therefore, we consider $L$-equivalence preservation in public and secret contexts *separately*. Then, we combine these individual results to prove $L$-equivalence preservation in arbitrary contexts, i.e., termination-insensitive non-interference. More precisely, we prove two preservation lemmas: the first relates executions in secret contexts ($pc \not\sqsubseteq L$) involving *arbitrary expressions*, while the other relates executions of the *same expression* in public contexts ($pc \sqsubseteq L$).

The first lemma ensures that programs cannot leak secret data implicitly through the *program control-flow*. For example, consider the program **if** $s$ **then** $e_1$ **else** $e_2$, which branches on a secret boolean $s$. Depending on the value of $s$, this program evaluates either expression $e_1$ or $e_2$, which may reveal the value of the secret through secret-dependent store updates (e.g., **if** $s$ **then** $p :=$ **true else** () for some public reference $p$) or results (e.g., **if** $s$ **then true else false**). This lemma ensures that even these programs cannot leak secrets through

the final result or observable changes to the stores. Formally, we have to show that programs preserve $L$-equivalence in secret contexts even if they evaluate different expressions, i.e., if $\Sigma_1 \approx_L \Sigma_2$, $\langle \Sigma_1, e_1 \rangle \Downarrow_H^{\theta_1} c_1$ and $\langle \Sigma_2, e_2 \rangle \Downarrow_H^{\theta_2} c_2$, then $c_1 \approx_L c_2$.

How should we prove this lemma? At first, we might try to prove it *directly* by induction on the reduction steps. However, this approach is not practical: since these executions involve *arbitrary* expressions, we would have to reason about a large number of completely unrelated reduction steps! Instead, we observe that these programs are executed in secret contexts ($pc = H$), so they are restricted by the security monitor to avoid leaks. From these restrictions, we establish two program *invariants* to show that the final stores and values are $L$-equivalent: in secret contexts, programs can (i) modify *only* secret memories (*Store Confinement*), and (ii) produce results labeled secret (Property 1). Notice that these invariants hold for individual executions: we still need to combine them to relate the final configurations of the two executions. To do that, we reason separately about the final stores and values. The final stores are related via a *Square Commutative Diagram* (Figure 2.6), while the secret results are trivially $L$-equivalent by rule [VALUE$_H$] (Figure 2.5).

**Lemma 2.3** ($L$-Equivalence Preservation in Secret Contexts)**.** For all program counter labels $pc_1 \not\sqsubseteq L$ and $pc_2 \not\sqsubseteq L$, and *arbitrary* expressions $e_1$ and $e_2$, if $\Sigma_1 \approx_L \Sigma_2$, $\langle \Sigma_1, e_1 \rangle \Downarrow_{pc_1}^{\theta_1} c_1$, and $\langle \Sigma_2, e_2 \rangle \Downarrow_{pc_2}^{\theta_2} c_2$, then $c_1 \approx_L c_2$.

*Proof.* Assume $pc_1 \not\sqsubseteq L$, $pc_2 \not\sqsubseteq L$, $\Sigma_1 \approx_L \Sigma_2$, and let the final configurations be $c_1 = \langle \Sigma_1', v_1 \rangle$ and $c_2 = \langle \Sigma_2', v_2 \rangle$. First, we apply *Store Confinement* (Lemma 2.2) to $\langle \Sigma_1, e_1 \rangle \Downarrow_{pc_1}^{\theta_1} \langle \Sigma_1', v_1 \rangle$ and $\langle \Sigma_2, e_2 \rangle \Downarrow_{pc_2}^{\theta_2} \langle \Sigma_2', v_2 \rangle$ and obtain $\Sigma_1 \approx_L \Sigma_1'$ and $\Sigma_2 \approx_L \Sigma_2'$, respectively. Then, we construct the *Square Commutative Diagram for Stores* (Lemma 2.1) using $\Sigma_1 \approx_L \Sigma_2$, $\Sigma_1' \approx_L \Sigma_1\ \Sigma_2 \approx_L \Sigma_2'$, and obtain $\Sigma_1' \approx_L \Sigma_2'$. To show that the values $v_1 = {r_1}^{\ell_1}$ and $v_2 = {r_2}^{\ell_2}$ are $L$-equivalent, it suffices to show that they are labeled *secret*. Since $pc_1 \not\sqsubseteq L$ and $pc_2 \not\sqsubseteq L$ by assumption, $pc_1 \sqsubseteq \ell_1$ and $pc_2 \sqsubseteq \ell_2$ by Property 1, we have $\ell_1 \not\sqsubseteq L$ and $\ell_2 \not\sqsubseteq L$, and thus $v_1 = {r_1}^{\ell_1} \approx_L {r_2}^{\ell_2} = v_2$ by rule [VALUE$_H$]. Since $\Sigma_1' \approx_L \Sigma_2'$ and $v_1 \approx_L v_2$, we have $c_1 = \langle \Sigma_1', v_1 \rangle \approx_L \langle \Sigma_2', v_2 \rangle = c_2$, as desired. □

We now turn our attention to $L$-equivalence preservation in *public contexts*. Unlike $L$-equivalence preservation in secret contexts, we must inspect the program executions and examine their specific reduction steps to prove this lemma. This is because these executions occur in a public context ($pc \sqsubseteq L$), therefore their side-effects and final results may be observable by the attacker and so we have to verify that no leaks occur in each case. Luckily, this task is simplified by the fact that the initial configurations are $L$-equivalent and thus contain $\alpha$-equivalent expressions. Intuitively, this means that the two programs are *synchronized* and proceed in *lock-step*. In other words, their reductions are almost identical, i.e., the programs perform the same operation on $L$-equivalent inputs and so produce $L$-equivalent outputs. This makes the proof of this lemma straightforward in most cases. However, reductions that involve control-flow expressions (e.g., $\mathbf{case}(e, x.e_1, x.e_2)$) are more complicated. In general, these programs may follow different paths and evaluate different expressions: how can they preserve $L$-equivalence? Intuitively, since the programs considered in this lemma are initially synchronized, they can start following different paths *only* after branching on secret data. As a result, their program counter label gets tainted and the programs enter a *secret context*, in which public side-effects are not allowed and results are guaranteed to be labeled secret, as proved in the previous lemma.

**Lemma 2.4** ($L$-Equivalence Preservation in Public Contexts)**.** For all program counter labels $pc \sqsubseteq L$, if $c_1 \approx_L c_2$, $\theta_1 \approx_L \theta_2$, $c_1 \Downarrow_{pc}^{\theta_1} c_1'$, and $c_2 \Downarrow_{pc}^{\theta_2} c_2'$, then $c_1' \approx_L c_2'$.

*Proof.* Let $c_1 = \langle \Sigma_1, e_1 \rangle$ and $c_2 = \langle \Sigma_2, e_2 \rangle$ and proceed by induction on the big-step reductions. Since $c_1 \approx_L c_2$, we know that the initial stores are $L$-equivalent, i.e., $\Sigma_1 \approx_L \Sigma_2$, and the big-step reductions evaluate $\alpha$-equivalent expressions, i.e., $e_1 \equiv_\alpha e_2$. As a result, the expressions step following the same rule in most cases and thus produce $L$-equivalent values and stores.

The interesting cases are those that influence the control-flow of the program, where the two executions may deviate from each other, i.e., case [Case$_1$], [Case$_2$], and [App]. In these cases, the security label $\ell$ of the scrutinee determines whether the two executions stay in a public

context ($\ell \sqsubseteq L$) and remain *synchronized* on the same path, or not ($\ell \not\sqsubseteq L$). For example, consider expression $\mathbf{case}(e, x.e_1, x.e_2)$, which steps through rule [CASE$_1$] in the first reduction and either through rule [CASE$_1$] or [CASE$_2$] in the second reduction (the opposite cases are symmetric). If both rules step through rule [CASE$_1$], then the scrutinees are both left injections, i.e., $\mathbf{inl}(v_1)^{\ell_1} \approx_L \mathbf{inl}(v_2)^{\ell_2}$ by induction hypothesis. Then, we perform case analysis on the $L$-equivalence judgment and have two sub-cases: [VALUE$_L$] and [VALUE$_H$]. In the first case, both scrutinees are public, i.e., $\ell_1 = \ell_2 \sqsubseteq L$ and $v_1 \approx_L v_2$ by rule [VALUE$_L$], and the proof follows by induction. In the other case, both scrutinees are secret, i.e., $\ell_1 \not\sqsubseteq L$ and $\ell_2 \not\sqsubseteq L$ by rule [VALUE$_H$], and the program enters a secret context and we apply *$L$-equivalence Preservation in Secret Contexts* (Lemma 2.3). Finally, if $\mathbf{case}(e, x.e_1, x.e_2)$ steps through different rules (e.g., [CASE$_1$] and [CASE$_2$]), then the scrutinees *must* be secret and thus we also apply *$L$-equivalence Preservation in Secret Contexts*. In particular, it is impossible for $L$-equivalent public scrutinees to have different injection. To see that, assume $\mathbf{inl}(v_1)^{\ell_1} \approx_L \mathbf{inr}(v_2)^{\ell_2}$ and $\ell_1 = \ell_2 \sqsubseteq L$ by rule [VALUE$_L$]. Then, we also have a proof that the *raw values* are $L$-equivalent, i.e., $\mathbf{inl}(v_1) \approx_L \mathbf{inr}(v_2)$. But this is impossible: raw values of sum type can be related only if they are the same injection (i.e., rules [INL] and [INR] in Figure 2.5). □

Finally, we combine the $L$-equivalence preservation lemmas from above and prove *termination-insensitive non-interference* (TINI) for $\lambda^{dFG}$.

**Theorem 1** ($\lambda^{dFG}$-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 case analysis over the program counter label $pc$. If $pc \sqsubseteq L$, we apply *$L$-equivalence Preservation in Public Contexts* (Lemma 2.4). If $pc \not\sqsubseteq L$, we apply *$L$-equivalence Preservation in Secret Contexts* (Lemma 2.3). □

## 2.3 Flow-Sensitive References

This section extends $\lambda^{dFG}$ with *flow-sensitive* references (Austin and Flanagan, 2009), an important feature to boost the permissiveness of

IFC systems. These references differ slightly from the labeled references presented in Section 2.1.2, which are instead *flow-insensitive*. The key difference between them lies in the way the IFC system treats their label. In particular, the label of flow-insensitive references is *immutable*, i.e., when a program creates a reference, the IFC monitor assigns it a label, which remains fixed throughout the execution of the program. In contrast, the label of flow-sensitive references is *mutable*. Intuitively, the label of these references can change to reflect the sensitivity of the data that they currently store. This change is completely transparent to the program: when the program writes some data to a flow-sensitive reference, the IFC monitor simply replaces the label of the reference with the label of the new content. However, if added naively, mutable labels open a new channel for implicitly leaking data, therefore, the IFC monitor changes the labels of flow-sensitive references only as long as this operation does not leak information.

In this section, we formally discuss the differences of flow-sensitive references and their subtleties for security in the context of $\lambda^{dFG}$. First, we show that flow-sensitive references are more permissive than flow-insensitive references with an example.

**Example 2.1.** Consider the following program, which creates a new reference, initially containing some public data $p$, overwrites it with secret data $s$, and finally reads it back from the reference.

$$\begin{array}{l}\textbf{let } r = \textbf{new}(p) \textbf{ in}\\ \quad r := s;\\ \quad !\, r\end{array}$$

The execution of the $\lambda^{dFG}$ program above with program counter label public, i.e., $pc = L$, environment $\theta = [p \mapsto \textbf{true}^{L}, s \mapsto \textbf{false}^{H}]$, and empty store $\Sigma = \lambda\ell.[\,]$ is aborted by the IFC monitor with flow-insensitive references from Section 2.1.2. Rule [New] (Figure 2.4) assigns the *fixed* public label $L$ to the reference, which then causes rule [Write] to fail, as the program tries to write secret data (i.e., $\textbf{false}^{H}$) into a public reference (i.e., $0_{L}$). In particular, the last premise of rule [Write] does not hold ($H \not\sqsubseteq L$) and thus the program gets stuck. Had the

monitor not aborted the execution, the program would have leaked the secret value of $s$. Specifically, the program would have written the *raw* value of $s$, i.e., **false**, into the public $L$-labeled memory, from where it would be extracted by the last instruction through rule [READ] and labeled with $L$, i.e., $\mathbf{false}^{L}$, causing the leak. In contrast, the IFC monitor presented in this section and extended with flow-sensitive references does *not* abort the program. The extended monitor allows the program to write secret data into the public reference, but after doing so, it *upgrades* the label of the reference from $L$ to $H$ to indicate that it now contains secret data. As a result, when the program reads back the reference in the last instruction, the result gets tainted with $H$, i.e., $\mathbf{false}^{H}$, thus eliminating the leak.

In the following, we extend $\lambda^{dFG}$ with flow-sensitive references (Section 2.3.1), define their operational semantics (Section 2.3.2), and finally reestablish non-interference (Section 2.3.3).

### 2.3.1 Syntax

We introduce the syntax of $\lambda^{dFG}$ extended with flow-sensitive references in Figure 2.8a, where we omit the constructs and the syntactic categories that remain unchanged. First, we annotate the type of references with flow-sensitivity tags, e.g., $\mathbf{Ref}\ s\ \tau$, where the tag $s$ allows programs to distinguish between flow-insensitive ($s = I$) and flow-sensitive ($s = S$) references. We do not introduce new constructs to create, write, and read flow-sensitive references. Instead, we reuse the same constructs introduced for flow-sensitive references (i.e., $\mathbf{new}(e)$, $e_1 := e_2$, $!e$, and $\mathbf{labelOfRef}(e)$ from Figure 2.1)[7] and rely on the type-level sensitivity tag to determine which kind of reference is used.[8] Raw values for flow-sensitive references are plain addresses $n \in \mathbb{N}$. Since the label of flow-sensitive references is *mutable* (as explained above), these addresses are *not* annotated with a label like flow-insensitive references (e.g., $n_\ell$) and hence do not point into a labeled memory in the partitioned

[7] We do not bother to introduce separate constructs because the translations of these constructs given in Sections 5 and 6 are the same for both kinds of references.

[8] In this presentation we assume that terms are implicitly well-typed. In our mechanized proofs, terms are explicitly and intrinsically well-typed.

$$\begin{aligned}
\text{Flow-sensitivity tags: } & s ::= I \mid S \\
\text{Types: } & \tau ::= \cdots \mid \mathbf{Ref}\ s\ \tau \\
\text{Raw Values: } & r ::= \cdots \mid n \\
\text{Heaps: } & \mu ::= [\,] \mid r^{\ell} : \mu \\
\text{Configurations: } & c ::= \langle \Sigma, \mu, e \rangle
\end{aligned}$$

**(a)** Syntax.

(New-FS)
$$\frac{\langle \Sigma, \mu, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mu', v\rangle \qquad n = |\mu'| \qquad \mu'' = \mu'[n \mapsto v]}{\langle \Sigma, \mu, \mathbf{new}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mu'', n^{pc}\rangle}$$

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

(LabelOfRef-FS)
$$\frac{\langle \Sigma, \mu, e\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mu', n^{\ell_1}\rangle \qquad \mu'[n] = r^{\ell_2}}{\langle \Sigma, \mu, \mathbf{labelOfRef}(e)\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mu', {\ell_2}^{\ell_1 \sqcup \ell_2}\rangle}$$

(Write-FS)
$$\frac{\begin{array}{c}\langle \Sigma, \mu, e_1\rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mu', n^{\ell}\rangle \qquad \langle \Sigma', \mu', e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', \mu'', {r_2}^{\ell_2}\rangle \\ \mu''[n] = {r_1}^{\ell_1} \qquad \ell \sqsubseteq \ell_1 \qquad \mu''' = \mu''[n \mapsto {r_2}^{\ell_2 \sqcup \ell}]\end{array}}{\langle \Sigma, \mu, e_1 := e_2\rangle \Downarrow^{\theta}_{pc} \langle \Sigma'', \mu''', ()^{pc}\rangle}$$

**(b)** Dynamics. The shaded constraint is the *no-sensitive upgrade* security check.

**Figure 2.8:** $\lambda^{dFG}$ extended with flow-sensitive references.

store. Instead, we store both the content and the label of flow-sensitive references in the new, linear heap $\mu$, which can contain data at different security levels. Specifically, a flow-sensitive reference $n$ of type **Ref** $S\ \tau$ points to the $n$-th cell of the heap $\mu$, i.e., $\mu[n] = r^{\ell}$, for some raw value $r$ of type $\tau$ at security level $\ell$, which also represents the label of the reference. Finally, we extend program configurations with a heap $\mu$, e.g., $c = \langle \Sigma, \mu, e \rangle$.

### 2.3.2 Dynamics

Figure 2.8b gives the semantics rules for the $\lambda^{dFG}$ constructs that operate on flow-sensitive references.[9] Rule [New-FS] allocates a new flow-sensitive reference in the heap at fresh address $n = |\mu'|$, where value $v$ gets stored, i.e., $\mu'[n \mapsto v]$. Similarly to rule [New] (Figure 2.4), this rule does not leak data through implicit flows because value $v$ has security level at least equal to the program counter label.[10] Rule [Read-FS] is also similar to rule [Read]. The rule reads reference $n$ at security level $\ell$ from the heap $\mu'$, i.e., $\mu'[n] = r^{\ell'}$, and upgrades the label of $r$ to $\ell \sqcup \ell'$ to reflect the fact that reading this particular reference depends on information at security level $\ell$. Rule [LabelOfRef-FS] retrieves the label of reference $n$ by reading the corresponding cell in the heap, i.e., $\mu'[n] = r^{\ell_2}$, and protects it with the label itself and taints it with the security level of the reference, i.e., ${\ell_2}^{\ell_1 \sqcup \ell_2}$. Although the label of values in the heap represent also the label of the reference, we cannot obtain exactly the label of the reference by reading the reference and then projecting the label of the value, i.e., in general $\mathbf{labelOfRef}(e) \not\equiv \mathbf{labelOf}(!e)$ for a flow-sensitive reference $e$. To see this, suppose $e$ evaluates to reference $n^{\ell_1}$ such that $\mu[n] = r^{\ell_2}$ for some raw value $r$. Then, expression $\mathbf{labelOfRef}(e)$ results in ${\ell_2}^{\ell_1 \sqcup \ell_2}$ through rule [LabelOfRef-FS], which is in general different from $(\ell_1 \sqcup \ell_2)^{\ell_1 \sqcup \ell_2}$ obtained from $\mathbf{labelOf}(!e)$ through rule [LabelOf] (Figure 2.3) applied to rule [Read-FS]. Hence, we need to define $\mathbf{labelOfRef}(\cdot)$ as a primitive construct of the calculus.

[9] The rules from Figure 2.2-2.4 are adapted by simply threading the heap through the intermediate evaluations.

[10] In particular, Property 1 holds also for $\lambda^{dFG}$ extended with flow-sensitive references.

Rule [WRITE-FS] updates flow-sensitive references but, in contrast to rule [WRITE], it also updates the label of the (flow-sensitive) reference. In particular, the rule updates reference $n$ at security level $\ell$ by writing value ${r_2}^{\ell_2}$ tainted with $\ell$ in the $n$-th cell of the heap, i.e., $\mu''[n \mapsto {r_2}^{\ell_2 \sqcup \ell}]$. Notice that the label of the updated reference, i.e., $\ell_2 \sqcup \ell$, depends *only* on the security level of the data written into the reference and of the reference that gets updated—this label can be completely different from the label of the reference before the update, i.e., $\ell_1$ from $\mu''[n] = {r_1}^{\ell_1}$. For example, the rule allows turning a public reference into a secret one (i.e., changing the label of the reference from $L$ to $H$) by writing secret data into it (like in Example 2.1). Perhaps more surprisingly, the rule also allows changing the label in the opposite direction (i.e., from $H$ to $L$). For example, it is possible to make a secret reference public by overwriting its content with public data (e.g., swap $p$ and $s$ in Example 2.1). However, rule [WRITE-FS] is not completely unrestricted: it includes a security check known as *no-sensitive upgrade* (NSU) to avoid leaking information through *implicit flows* (Austin and Flanagan, 2009). Intuitively, this check ( shaded in Figure 2.8b) prohibits updates in which the decision to update a particular reference depends on information that is *more sensitive* than the label of the reference itself. The following example shows why this check is needed to avoid leaks.

**Example 2.2.** Consider the following program, which branches on secret data, updates an initially public (flow-sensitive) reference in one branch, and then reads back the reference.

$$\begin{array}{l}\textbf{let } r = \textbf{new}(p) \textbf{ in}\\ \quad \textbf{if } s \textbf{ then } r := s \textbf{ else } ();\\ \quad !\, r\end{array}$$

Without the no-sensitive upgrade check described above, this program *leaks* information through an implicit flow. Formally, we show that the program above breaks non-interference (Theorem 1). Consider two executions of the program with program counter label public, i.e., $pc = L$, and with two $L$-equivalent input environments, e.g., $\theta_1 = [p \mapsto \textbf{true}^L, s \mapsto \textbf{false}^H] \approx_L [p \mapsto \textbf{true}^L, s \mapsto \textbf{true}^H] = \theta_2$. In the first

execution with environment $\theta_1$, reference $r$ is *not* updated, and thus the result of the program is public value $p = \mathbf{true}^L$. Instead, in the second execution with environment $\theta_2$, the program updates reference $r$ (rule [WRITE-FS] without the NSU check) and thus we obtain the secret value $s = \mathbf{true}^H$ as result. However, this breaks Theorem 1: the input environments are $L$-equivalent, i.e., $\theta_1 \approx_L \theta_2$, but the results of the program are not, i.e., $p = \mathbf{true}^L \not\approx_L \mathbf{true}^H = s$, because $L \sqsubseteq L$, but $H \not\sqsubseteq L$ (neither rule [VALUE$_H$] nor [VALUE$_L$] from Figure 2.5 apply).[11]

To avoid such leaks, we include the no-sensitive upgrade check in rule [WRITE-FS] through the constraint $\ell \sqsubseteq \ell_1$, which ensures that the decision of updating reference $n$ depends on information at some security level $\ell$ below the current label of the reference, i.e., $\ell_1$ from $\mu''[n] = {r_1}^{\ell_1}$. This constraint causes the IFC monitor to abort the second execution of the program above ($H \not\sqsubseteq L$), which gets stuck and thus satisfies (trivially) Theorem 1.

### 2.3.3 Security

$\lambda^{dFG}$ extended with flow-sensitive references is also secure, i.e., it satisfies termination insensitive non-interference. However, the theorem for the extended calculus is more complicated than Theorem 1: it requires a relaxed notion of $L$-equivalence up to a *bijection*, which relates observable flow-sensitive references with different heap addresses, and a side-condition on program configurations to rule out dangling references (Banerjee and Naumann, 2005). In this section, we make these differences rigorous and we reestablish non-interference.

**The need for Bijections.** Since the label of flow-sensitive references can change throughout program execution, our semantics allocates cells for both public and secret references in a single, unlabeled heap. An unfortunate consequence of this design choice is that allocations made in secret contexts can influence the addresses of public references allocated in the rest of the program, which get shifted by the number of secret

[11] The mismatch in the label of the results can also be propagated to raw values through the label introspection primitives. We refer to (Austin and Flanagan, 2009) for a more elaborated example that leaks directly through raw values.

references previously allocated. This can be problematic for security because attackers can exploit these deterministic side-effects to leak information through the address of public references (Hedin and Sands, 2006). However, the addresses considered in this work are *opaque*, i.e., programs cannot inspect and compare them, and so attackers cannot use them to leak secrets. To show that, we extend the $L$-equivalence relation from Section 2.2 with a *bijection* (Banerjee and Naumann, 2005), a one-to-one correspondence between heap addresses, which we use to relate the addresses of references allocated in public contexts.

**Definition 1** (Bijection). A bijection $\beta : Addr \rightharpoonup Addr$ is a bijective finite partial function between heap addresses. Formally, for all addresses $n$ and $n'$, $\beta(n) = n' \iff \beta^{-1}(n') = n$, where $\beta^{-1}$ is the inverse function of $\beta$.

**Notation and Auxiliary Definitions.** In the following, we treat partial functions as sets of input-output pairs and for clarity we write $(n_1, n_2) \in \beta$ for $\beta(n_1) = n_2$, i.e., if the partial function $\beta$ is defined on input $n_1$ and has output $n_2$. We say that a bijection $\beta'$ *extends* another bijection $\beta$, written $\beta \subseteq \beta'$, if and only if for all input-output pairs $(n_1, n_2)$, $(n_1, n_2) \in \beta$ implies $(n_1, n_2) \in \beta'$. Bijection composition is written $\beta_2 \circ \beta_1 = \{(n_1, n_3) \mid (n_1, n_2) \in \beta_1 \wedge (n_2, n_3) \in \beta_2\}$ and the inverse operator as $\beta^{-1} = \{(n_2, n_1) \mid (n_1, n_2) \in \beta\}$. We also define the domain and range of bijections in the obvious way, i.e., $dom(\beta) = \{n_1 \mid (n_1, n_2) \in \beta\}$, and $rng(\beta) = \{n_2 \mid (n_1, n_2) \in \beta\}$. Furthermore, we write $\iota_n$ for the *finite*, partial identity bijection defined up to $n$, i.e., $\iota_n = \{(n', n') \mid n' \in \{0, \ldots, n-1\}\}$. Identity bijections satisfy the following laws.

**Property 3** (Identity Laws). For all bijections $\beta$ and $n \in \mathbb{N}$:

1. **Inverse Identity.** $\iota_n^{-1} \equiv \iota_n$.
2. **Absorb Left.** If $rng(\beta) \subseteq dom(\iota_n)$, then $\iota_n \circ \beta \equiv \beta$.
3. **Absorb Right.** If $dom(\beta) \subseteq rng(\iota_n)$, then $\beta \circ \iota_n \equiv \beta$.

**$L$-Equivalence up to Bijection.** Formally, we redefine $L$-equivalence as a relation $\approx_L^\beta \subseteq \mathit{Value} \times \mathit{Value}$ and write $v_1 \approx_L^\beta v_2$ to indicate that values $v_1$ and $v_2$ are indistinguishable to an attacker at security level $L$ up to bijection $\beta$, which relates the heap addresses of corresponding observable flow-sensitive references in $v_1$ and $v_2$. We extend the $L$-equivalence relation for all other syntactic categories (raw values, memories, and stores) with a bijection in the same way. Besides the bijection $\beta$, which we only add to $\approx_L$, all the rules from Figure 2.5 remain unchanged. In particular, rules [Ref$_L$] and [Ref$_H$] for flow-insensitive references simply ignore the bijection. Since we keep labeled memories partitioned in the store and isolated from the heap, the problem of mismatching addresses does not arise for flow-insensitive references. Intuitively, the *address space* of public memories is completely independent from the address space of secret memories, and thus allocations made in secret contexts cannot influence the memory addresses of public references. As a result, rule [Ref$_L$] guarantees that $L$-equivalent public references have exactly the same *memory address*. In contrast, heap allocations made in secret contexts can shift the *heap addresses* of observable flow-sensitive references (as explained above), which then may not be necessarily the same. Therefore, we relate observable flow-sensitive references with possibly different addresses through the following new rule:

$$\frac{(n_1, n_2) \in \beta}{n_1 \approx_L^\beta n_2} \text{(Ref-FS)}$$

Rule [Ref-FS] relates references with arbitrary heap addresses $n_1$ and $n_2$, as long as their addresses are "matched" by the bijection $\beta$, i.e., $(n_1, n_2) \in \beta$. Similarly, we define heap $L$-equivalence by relating heap cells that are at corresponding addresses according to the bijection.

**Definition 2** (Heap $L$-equivalence)**.** Two heaps $\mu_1$ and $\mu_2$ are $L$-equivalent up to bijection $\beta$, written $\mu_1 \approx_L^\beta \mu_2$, if and only if:

1. $dom(\beta) \subseteq \{0, \ldots, |\mu_1| - 1\}$,
2. $rng(\beta) \subseteq \{0, \ldots, |\mu_2| - 1\}$, and
3. For all addresses $n_1$ and $n_2$, if $(n_1, n_2) \in \beta$, then $\mu_1[n_1] \approx_L^\beta \mu_2[n_2]$.

In the definition above, the first two conditions require that the domain and the range of $\beta$ are contained in the address space of $\mu_1$ and $\mu_2$, respectively.[12] These side-conditions ensure that the heap addresses considered in the third condition are *valid*, i.e., they point to some cell in each heap. The definition of $L$-equivalence for heaps is complicated by the fact that heaps can contain both public and secret cells, i.e., cells allocated in public and secret contexts, respectively. In particular, the $L$-equivalence relation needs to relate the values contained in corresponding public cells, which are at possibly different addresses in the heaps.[13] This task is simplified by the bijection $\beta$, which identifies exactly the addresses of corresponding public cells. Therefore, the third condition above ensures that the values contained in the cells at addresses related by the bijection $\beta$, i.e., $(n_1, n_2) \in \beta$, are themselves $L$-equivalent, i.e., $\mu_1[n_1] \approx_L^\beta \mu_2[n_2]$. Finally, we extend $L$-equivalence for configurations with a bijection and additionally require their heaps to be related. Formally, $c_1 \approx_L^\beta c_2$ iff $c_1 = \langle \Sigma_1, \mu_1, v_1 \rangle$, $c_2 = \langle \Sigma_2, \mu_2, v_2 \rangle$, and all their components are related, i.e., $\Sigma_1 \approx_L^\beta \Sigma_2$, $\mu_1 \approx_L^\beta \mu_2$, and $v_1 \approx_L^\beta v_2$.

Next, we discuss some side-conditions over program configurations and inputs that are needed for reasoning with $L$-equivalence relations up to bijection.

**Valid References.** In Section 2.2, we proved that the $L$-equivalence relation is reflexive, symmetric, and transitive, and showed how these properties helped us structure and simplify the proof of non-interference. Does the $L$-equivalence extended with bijection also enjoy these properties? Unfortunately, reflexivity does not hold *unconditionally* anymore for $L$-equivalence relations up to bijection. The culprit of the problem is in the definition of $L$-equivalence for heaps. In this case, proving reflexivity requires showing that any heap is $L$-equivalent to itself up to the identity bijection defined over its address space, i.e., $\forall \mu.\mu \approx_L^{\iota_{|\mu|}} \mu$. Technically, this property does not hold in general for any *arbitrary* heap $\mu$. To see this, consider for example the ill-formed heap $\mu' = 1^L : [\,]$, which contains the *dangling* flow-sensitive reference $1^L$. In order to

[12] Heap addresses start at 0 just like memory addresses.

[13] Secret cells in each heap are simply disregarded as they may not necessarily have a counterpart in the other heap.

prove $\mu' \approx_L^{\iota_1} \mu'$, we must show that this dangling reference is $L$-equivalent to itself up to bijection $\iota_1$ (Definition 2.3), i.e., $1^L \approx_L^{\iota_1} 1^L$, which by rule [Ref-FS] would require a proof for the false statement $(1,1) \in \iota_1 = \{(0,0)\}$. One may be tempted to repair reflexivity by choosing a sufficiently large domain for the identity bijection, so that also dangling references can be related (e.g., for $\mu'$ we could pick $\iota_2$ and prove $1^L \approx_L^{\iota_2} 1^L$). Unfortunately, this is not possible without breaking the first and the second condition of Definition 2. For any heap $\mu$, the identity bijection $\iota_{|\mu|}$ has the largest domain and range that satisfies the side-conditions $dom(\iota_{|\mu|}) \subseteq \{0\,..\,|\mu|-1\}$ (Definition 2.1) and $rng(\iota_{|\mu|}) \subseteq \{0\,..\,|\mu|-1\}$ (Definition 2.2).

The solution to this technical nuisance is to simply prove a weaker version of reflexivity, which holds only for *well-formed* heaps (and similarly values, stores, memories, and configurations) that contain only *valid* (i.e., not dangling) references. Luckily, this restriction is inconsequential in our setting because references are always valid in $\lambda^{dFG}$. Intuitively, references are *unforgeable*: programs can only create fresh, valid references through the $\mathbf{new}(\cdot)$ construct. After creation references simply remain valid throughout program execution: they can be accessed, passed around, or stored, but never deallocated (we simply provide no construct to do so) or tempered. We now formalize this insight in the form of a *valid judgment*, which rules out dangling reference from all the categories of the calculus together with the proof that this judgment is an *invariant* of the operational semantics of $\lambda^{dFG}$.

**Valid Judgment.** Figure 2.9 defines a valid judgment for $\lambda^{dFG}$ values and configurations. These judgments ensure that all flow-sensitive references values of a program configuration are valid (i.e., not dangling) in a heap of a given size. For values (Figure 2.9a), this judgment takes the form of $n \vdash \mathbf{Valid}(v)$, which indicates that all heap addresses contained in value $v$ are valid in a heap of size $n$. This judgment is mutually defined with similar judgments for raw values, i.e., $n \vdash \mathbf{Valid}(r)$, and environments $n \vdash \mathbf{Valid}(\theta)$. Importantly, rule [Valid-FS-Ref] disallows *dangling references*: a flow-sensitive reference is valid if and only if its heap address $n'$ is strictly smaller than the size $n$ of the heap. Notice that this judgment holds unconditionally for *flow-insensitive*

values through rule [Valid-FI-Ref]. Intuitively, these references contain memory addresses, which are not affected by the technical issue described above. Most of the remaining rules are fairly straightforward: they simply require that all sub-values are homogeneously valid in heaps of the same size. For example, values $r^\ell$ is valid only if and only if the raw value $r$ is also valid (rule [Valid-Value]) and closures $(x.e, \theta)$ are valid if and only if the value environment $\theta$ is also valid (rule [Valid-Closure]).[14] Environments $\theta$ are valid if they contain only valid values, i.e., $n \vdash \mathbf{Valid}(\theta)$ iff $\forall x \in dom(\theta).n \vdash \mathbf{Valid}(\theta(x))$. Similarly, valid stores $\Sigma$ contain only valid memories, i.e., $n \vdash \mathbf{Valid}(\Sigma)$ iff $\forall \ell \in \mathscr{L}.n \vdash \mathbf{Valid}(\Sigma(\ell))$, and so on for memories and heaps, i.e., $n \vdash \mathbf{Valid}(M)$ iff $\forall n \in \{0, \ldots, |M| - 1\}.n \vdash \mathbf{Valid}(M[n])$ and $n \vdash \mathbf{Valid}(\mu)$ iff $\forall n \in \{0, \ldots, |\mu| - 1\}.n \vdash \mathbf{Valid}(\mu[n])$, respectively. Finally, the judgments for initial and final configurations (Figure 2.9b) instantiate the parameter $n$ of the judgments above with the size of the heap. For example, a configuration $c = \langle \Sigma, \mu, e \rangle$ and input environment $\theta$ are valid through rule [Valid-Inputs] if and only if all the components of the configuration and the value environment are valid in the heap of size $n = |\mu|$, i.e., $\vdash \mathbf{Valid}(c, \theta)$ iff $n \vdash \mathbf{Valid}(\Sigma)$, $n \vdash \mathbf{Valid}(\mu)$, and $n \vdash \mathbf{Valid}(\theta)$. Rule [Valid-Final] is for final configurations and is similar.

Before showing that this valid judgment is an invariant of the semantics of $\lambda^{dFG}$, we prove two simple helping lemmas. The first lemma shows that values that are valid in a heap of a certain size, are also valid in any larger heap, while the second lemma shows that heaps can only grow larger in the semantics of $\lambda^{dFG}$.

**Lemma 2.5** (Valid Weakening)**.** For all values $v$, environments $\theta$, raw values $r$ and natural numbers $n$ and $n'$:

1. If $n \vdash \mathbf{Valid}(v)$ and $n \leqslant n'$, then $n' \vdash \mathbf{Valid}(v)$,
2. If $n \vdash \mathbf{Valid}(r)$ and $n \leqslant n'$, then $n' \vdash \mathbf{Valid}(r)$
3. If $n \vdash \mathbf{Valid}(\theta)$ and $n \leqslant n'$, then $n' \vdash \mathbf{Valid}(\theta)$

[14] Since expressions do not contain heap addresses or values (they belong to distinct syntactic categories), we do not need to define a valid judgment for expressions.

$$\frac{n \vdash \mathbf{Valid}(r)}{n \vdash \mathbf{Valid}(r^{\ell})}\ (\textsc{Valid-Value}) \qquad \frac{n' < n}{n \vdash \mathbf{Valid}(n')}\ (\textsc{Valid-FS-Ref}) \qquad n \vdash \mathbf{Valid}(n'_{\ell})\ (\textsc{Valid-FI-Ref})$$

$$\frac{n \vdash \mathbf{Valid}(\theta)}{n \vdash \mathbf{Valid}((x.e, \theta))}\ (\textsc{Valid-Closure})$$

**(a)** Judgment for values $n \vdash \mathbf{Valid}(v)$ and raw values $n \vdash \mathbf{Valid}(r)$ (selected cases).

$$\frac{n = |\mu| \qquad c = \langle \Sigma, \mu, e \rangle \qquad n \vdash \mathbf{Valid}(\Sigma) \qquad n \vdash \mathbf{Valid}(\mu) \qquad n \vdash \mathbf{Valid}(\theta)}{\vdash \mathbf{Valid}(c, \theta)}\ (\textsc{Valid-Inputs})$$

$$\frac{n = |\mu| \qquad c = \langle \Sigma, \mu, v \rangle \qquad n \vdash \mathbf{Valid}(\Sigma) \qquad n \vdash \mathbf{Valid}(\mu) \qquad n \vdash \mathbf{Valid}(v)}{\vdash \mathbf{Valid}(c)}\ (\textsc{Valid-Outputs})$$

**(b)** Judgments for valid program inputs and outputs in $\lambda^{dFG}$.

**Figure 2.9:** Some valid judgments for $\lambda^{dFG}$.

*Proof.* By mutual induction over the judgments and using transitivity of $\leqslant$ for the case [Valid-FS-Ref]. □

**Lemma 2.6** (Heaps Only Grow)**.** Let $c = \langle \Sigma, \mu, e \rangle$, $c' = \langle \Sigma', \mu', v \rangle$, if $c \Downarrow^{\theta}_{pc} c'$, then $|\mu| \leqslant |\mu'|$.

*Proof.* By induction on the big-step reduction. In particular, notice that no rule deallocates cells from the heap, which can only grow through rule [New-FS]. □

**Property 4** (Valid Invariant)**.** If $c \Downarrow^{\theta}_{pc} c'$ and $\vdash \mathbf{Valid}(c, \theta)$, then $\vdash \mathbf{Valid}(c')$.

*Proof.* By induction on the big-step reduction and using Lemmas 2.5 and 2.6. □

We now reconsider the properties of $L$-equivalence up to bijection.

**Property 5.** For all values, raw values, environments, memories, and stores $x$, $y$, $z$, and $n \in \mathbb{N}$:

1. **Restricted Reflexivity.** If $n \vdash \mathbf{Valid}(x)$, then $x \approx_L^{\iota_n} x$.
2. **Symmetricity.** If $x \approx_L^{\beta} y$, then $y \approx_L^{\beta^{-1}} x$.
3. **Transitivity.** If $x \approx_L^{\beta_1} y$ and $y \approx_L^{\beta_2} z$, then $x \approx_L^{\beta_2 \circ \beta_1} z$.
4. **Weakening.** If $x \approx_L^{\beta} y$ and $\beta \subseteq \beta'$, then $x \approx_L^{\beta'} y$.

First, *reflexivity* up to a *identity bijection* $\iota_n$ is restricted to program constructs that are valid in a heap of size $n$, as explained above. In a relation $x \approx_L^{\beta} y$, the bijection $\beta$ maps the heap addresses of $x$ to the corresponding heap addresses of $y$. Then, in order to obtain the mapping from $y$ to $x$, the bijection must be inverted in the symmetric relation, i.e., $y \approx_L^{\beta^{-1}} x$. Similarly, when composing $x \approx_L^{\beta_1} y$ and $y \approx_L^{\beta_2} z$ through *transitivity*, the bijections must also be composed, i.e., $x \approx_L^{\beta_2 \circ \beta_1} z$, so that the composed bijection $\beta_2 \circ \beta_1$ first maps the addresses of $x$ to the corresponding addresses of $y$ through $\beta_1$, and then the addresses of $y$ to those of $z$ through $\beta_2$. Lastly, *weakening* allows to relax a relation $x \approx_L^{\beta} y$ with an extended bijection $\beta' \supseteq \beta$. All the properties above hold also for $L$-equivalence for heaps (Definition 2), with the exception for *weakening*, due to the side conditions on the domain and range of the bijection.

We conclude this part with the *square commutative digram* for heaps, outlined in Figure 2.10.[15] In the diagram, the arrows connect $L$-equivalent heaps up to a given bijection, e.g., the arrow from $\mu_1$ to $\mu_1'$ labeled $\beta_1$ indicates that $\mu_1 \approx_L^{\beta_1} \mu_1'$. Compared to the square diagram without bijections from Figure 2.6, this diagram is complicated by the fact that the bijections must be composed and inverted appropriately, as needed by *symmetricity* and *transitivity*, in order to obtain the

[15] We omit the analogous square commutative diagram for stores.

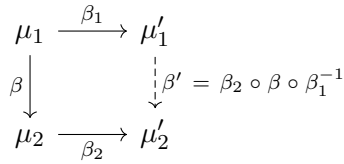


**Figure 2.10:** Square commutative diagram for heaps (Lemma 2.7). Solid arrows represent the assumptions of the lemma and the dashed arrow represents the conclusion. If $\beta_1 = \iota_{|\mu_1|}$, $\beta_2 = \iota_{|\mu_2|}$, then $\beta'$ is simply equal to $\beta$.

dashed arrow that completes the square. However, when $\beta_1$ and $\beta_2$ are (appropriate) identity bijections, the bijection between the final heaps can be simplified and shown to be equal to the bijection between the initial heaps.

**Lemma 2.7** (Square Commutative Diagram for Heaps)**.** If $\mu_1 \approx_L^{\beta} \mu_2$, $\mu_1 \approx_L^{\beta_1} \mu_1'$, and $\mu_2 \approx_L^{\beta_2} \mu_2'$, then $\mu_1' \approx_L^{\beta'} \mu_2'$, where $\beta' = \beta_2 \circ \beta \circ \beta_1^{-1}$. Furthermore, if $\beta_1 = \iota_{|\mu_1|}$ and $\beta_2 = \iota_{|\mu_2|}$, then $\mu_1' \approx_L^{\beta} \mu_2'$.

*Proof.* By applying *symmetricity* (Property 5.2) and *transitivity* (Property 5.3) like in Lemma 2.1, we obtain directly $\mu_1' \approx_L^{\beta'} \mu_2'$ and $\beta' = \beta_2 \circ \beta \circ \beta_1^{-1}$. If $\beta_1 = \iota_{|\mu_1|}$ and $\beta_2 = \iota_{|\mu_2|}$, then $\beta' = \iota_{|\mu_2|} \circ \beta \circ \iota_{|\mu_1|}^{-1}$. Using the *identity laws* (Property 3), we show that $\beta' = \iota_{|\mu_2|} \circ \beta \circ \iota_{|\mu_1|}^{-1} = \beta$ and thus we have $\mu_1' \approx_L^{\beta} \mu_2'$. □

**Termination-Insensitive Non-Interference.** We now turn our attention to the termination-insensitive non-interference theorem for $\lambda^{dFG}$ extended with flow-sensitive references. First, the theorem assumes that the initial configurations are $L$-equivalent up to a bijection, which we use to account for secret-dependent heap allocations and hence relate the heap addresses of corresponding public flow-sensitive references, as explained above. Then, the theorem guarantees that the final configurations are also $L$-equivalent, but up to some *extended bijection*. Intuitively, programs may dynamically allocate new cells in the heap, therefore the heap addresses contained in their final configurations must be shown to be related with respect to the address space of the final heaps. Additionally, the theorem explicitly assumes that the initial

program configurations are valid and thus do not contain dangling references. As explained before, this extra assumption is needed for technical reasons and, besides having to explicitly propagate it as needed in our mechanized proof scripts, it does not impact the security guarantees of $\lambda^{dFG}$. The proof of this theorem is also structured on two key lemmas: *store and heap confinement* and $L$*-equivalence preservation*. In the following, we focus on the changes needed to adapt these lemmas for flow-sensitive references, heaps, and $L$-equivalence up to bijection.

Confinement guarantees that programs running in a secret context cannot leak secret data implicitly through the program state, i.e., the store and the heap. Formally, we show that if the program counter label is above the attacker's label ($pc \not\sqsubseteq L$), then the initial store and heap are $L$-equivalent to the final store and heap obtained at the end of the execution, up to the *identity bijection*. In the lemma, $L$-equivalence is up to the identity bijection defined over the address space of the initial heap. Intuitively, this is because the references allocated in secret contexts cannot be observed by the attacker, and therefore must be disregarded by the bijection, which instead keeps track only of observable references, i.e., those allocated in public contexts.

**Lemma 2.8** (Store and Heap Confinement)**.** For all configurations $c = \langle \Sigma, \mu, e \rangle$, $c' = \langle \Sigma', \mu', v \rangle$, program counter labels $pc \not\sqsubseteq L$, if $\vdash \mathbf{Valid}(c, \theta)$ and $c \Downarrow^{\theta}_{pc} c'$, then $\Sigma \approx_L^{\iota_{|\mu|}} \Sigma'$ and $\mu \approx_L^{\iota_{|\mu|}} \mu'$.

*Proof.* By induction on the big-step reduction and using the fact the semantics preserves valid references (Property 4) to propagate the valid judgment through the intermediate configurations. In particular, we apply *restricted reflexivity* (Property 5.1) in the base cases, and *transitivity* (Property 5.3) and the bijection *identity laws* (Property 3) in the inductive cases. Intuitively, when we apply transitivity, we need to show that the composition of identity bijections defined over different heaps results in the identity bijection over the initial heap required by the lemma. For example, to show that $\mu \approx_L^{\iota_{|\mu|}} \mu''$ through some intermediate heap $\mu'$ such that $\mu \approx_L^{\iota_{|\mu|}} \mu'$ and $\mu' \approx_L^{\iota_{|\mu'|}} \mu''$, we apply transitivity and obtain $\mu \approx_L^{\iota_{|\mu'|} \circ \iota_{|\mu|}} \mu''$. Since heaps only grow in size (Lemma 2.6), we have that $|\mu| \leqslant |\mu'|$, thus $rng(\iota_{|\mu|}) \subseteq dom(\iota_{|\mu'|})$

and $\iota_{|\mu'|} \circ \iota_{|\mu|} = \iota_{|\mu|}$ by Property 3.2 and so we have $\mu \approx_L^{\iota_{|\mu|}} \mu''$, as required. □

For $L$-equivalence preservation, we assume that the program inputs are $L$-equivalent up to some bijection $\beta$ and must show that the final configurations are $L$-equivalent up to some extended bijection $\beta' \supseteq \beta$. Since we cannot predict, in general, how arbitrary programs allocate references at run-time, in these lemmas we *existentially quantify* the bijection $\beta'$ that relates the heap addresses in final configurations, but guarantee that the final bijection $\beta'$ is at least as large as the initial bijection $\beta$, i.e., $\beta \subseteq \beta'$. This invariant is critical for the proof, where the inductive steps require combining $L$-equivalent values and environments with different bijections. In particular, the fact that bijections only get extended, e.g., $\beta \subseteq \beta'$, lets us lift $L$-equivalence relations up to a small bijection $\beta$ into $L$-equivalence relations up to a larger bijection $\beta'$ via *weakening* (Property 5.4).

Like before, we prove two $L$-equivalence preservation lemmas, for secret and public contexts, respectively, which we then combine in the proof of termination-insensitive non-interference. For preservation in secret contexts, we observe that the bijection that relates the final configurations is *exactly the same* bijection that relates the initial configurations.[16] Since the program executions occur in secret contexts, their allocations are secret-dependent and therefore not observable by the attacker, which is reflected in the lemma by relating the initial and final configurations with the same bijection. The proof technique for this lemma is similar to that of the previous $L$-equivalence preservation lemma. We consider each program execution individually and reason independently about program state (i.e., heap and store) and values in the final configurations. In particular, we apply *store and heap confinement* to each execution and relate the initial and final program states with the *identity bijection*, which are then combined in a square commutative diagram (Figure 2.10) to relate the final configurations.

[16] This is a special case of $L$-equivalence preservation, where we can predict the final bijection $\beta$, and therefore we omit the existential quantification. Notice that in this case the invariant $\beta \subseteq \beta$ is trivially satisfied.

**Lemma 2.9** ($L$-Equivalence Preservation in Secret Contexts)**.** For all program counter labels $pc_1 \not\sqsubseteq L$ and $pc_2 \not\sqsubseteq L$, valid inputs $\vdash \mathbf{Valid}(c_1, \theta_1)$ and $\vdash \mathbf{Valid}(c_2, \theta_2)$, such that $c_1 = \langle \Sigma_1, \mu_1, e_1 \rangle$, $c_2 = \langle \Sigma_2, \mu_2, e_2 \rangle$, $\Sigma_1 \approx^{\beta}_{L} \Sigma_2$ and $\mu_1 \approx^{\beta}_{L} \mu_2$, if $\langle \Sigma_1, \mu_1, e_1 \rangle \Downarrow^{\theta_1}_{pc_1} c_1$ and $\langle \Sigma_2, \mu_2, e_2 \rangle \Downarrow^{\theta_2}_{pc_2} c_2$, then $c_1 \approx^{\beta}_{L} c_2$.

*Proof.* Assume $pc_1 \not\sqsubseteq L$, $pc_2 \not\sqsubseteq L$, $\Sigma_1 \approx^{\beta}_{L} \Sigma_2$, $\mu_1 \approx^{\beta}_{L} \mu_2$, and let the final configurations be $c_1 = \langle \Sigma_1', \mu_1', v_1 \rangle$ and $c_2 = \langle \Sigma_2', \mu_2', v_2 \rangle$. First, we apply *Store and Heap Confinement* (Lemma 2.8) to $\langle \Sigma_1, \mu_1, e_1 \rangle \Downarrow^{\theta_1}_{pc_1} \langle \Sigma_1', \mu_1', v_1 \rangle$ and $\langle \Sigma_2, \mu_2, e_2 \rangle \Downarrow^{\theta_2}_{pc_2} \langle \Sigma_2', \mu_2, v_2 \rangle$ and obtain $\mu_1 \approx^{\iota_{|\mu_2|}}_{L} \mu_1'$ and $\mu_2 \approx^{\iota_{|\mu_2|}}_{L} \mu_2'$, respectively. Then, we construct the *Square Commutative Diagram for Heaps* (Lemma 2.7) using $\mu_1 \approx^{\beta}_{L} \mu_2$, $\mu_1 \approx^{\iota_{|\mu_1|}}_{L} \mu_1'$ and $\mu_2 \approx^{\iota_{|\mu_2|}}_{L} \mu_2'$, and obtain $\mu_1' \approx^{\beta}_{L} \mu_2'$. We derive $\Sigma_1' \approx^{\beta}_{L} \Sigma_2'$ with a similar construction and prove $v_1 \approx^{\beta}_{L} v_2$ like in Lemma 2.3. □

The proof of $L$-equivalence preservation in public contexts also follows the same structure of the proof without heaps. In particular, the proof is by simultaneous induction on the two big-step reductions and relies on *$L$-Equivalence Preservation in Secret Contexts* when the program executions deviate from each other at a secret-dependent control-flow point. Here, we discuss only the interesting case where the programs allocate a flow-sensitive reference and we must extend the bijection with a new pair of matching addresses.

**Lemma 2.10** ($L$-Equivalence Preservation in Public Contexts)**.** For all program counter labels $pc \sqsubseteq L$, valid inputs $\vdash \mathbf{Valid}(c_1, \theta_1)$ and $\vdash \mathbf{Valid}(c_2, \theta_2)$, such that $c_1 \approx^{\beta}_{L} c_2$, $\theta_1 \approx^{\beta}_{L} \theta_2$, if $c_1 \Downarrow^{\theta_1}_{pc} c_1'$, and $c_2 \Downarrow^{\theta_2}_{pc} c_2'$, then there exists an extended bijection $\beta' \supseteq \beta$, such that $c_1' \approx^{\beta'}_{L} c_2'$.

*Proof.* From $L$-equivalence, i.e., $c_1 \approx^{\beta}_{L} c_2$, the program expressions are $\alpha$-equivalent, therefore the two reductions step following the same rule and produce $L$-equivalent results and perform $L$-equivalent operations on the program store and heap. In particular, for rule [New-FS], the programs allocate $L$-equivalent heap cells, e.g., $v_1 \approx^{\beta'}_{L} v_2$ for some extended bijection $\beta' \supseteq \beta$ by induction hypothesis, at fresh addresses $n_1 = |\mu_1'|$ and $n_2 = |\mu_2'|$ of related heaps $\mu_1' \approx^{\beta'}_{L} \mu_2'$, which remain

related under the *extended* bijection $\beta'' = \beta' \cup \{(n_1, n_2)\} \supseteq \beta$ by transitivity, i.e., $\mu_1[n_1 \mapsto v_1] \approx_L^{\beta''} \mu_2[n_2 \mapsto v_2]$, and so produce related references ${n_1}^L \approx_L^{\beta''} {n_2}^L$ by rule [VALUE$_L$] applied to rule [REF-FS]. □

Lastly, we combine the $L$-equivalence preservation lemmas for public and secret contexts and prove termination-insensitive non-interference for $\lambda^{dFG}$ extended with flow-sensitive references.

**Theorem 2** ($\lambda^{dFG}$-TINI with Bijections)**.** For all program counter labels $pc$ and valid inputs $\vdash \mathbf{Valid}(c_1, \theta_1)$ and $\vdash \mathbf{Valid}(c_2, \theta_2)$, such that $c_1 \approx_L^{\beta} c_2$, $\theta_1 \approx_L^{\beta} \theta_2$, if $c_1 \Downarrow_{pc}^{\theta_1} c_1'$, and $c_2 \Downarrow_{pc}^{\theta_2} c_2'$, then there exists an extended bijection $\beta' \supseteq \beta$, such that $c_1' \approx_L^{\beta'} c_2'$.

**Conclusion.** Dynamic language-based fine-grained IFC, of which $\lambda^{dFG}$ 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 IFC 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.

$$
\begin{aligned}
\text{Types: } & \tau ::= \mathbf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mathscr{L} \\
& \mid \mathbf{LIO}\ \tau \mid \mathbf{Labeled}\ \tau \mid \mathbf{Ref}\ \tau \\
\text{Labels: } & \ell, pc \in \mathscr{L} \\
\text{Addresses: } & n \in \mathbb{N} \\
\text{Environments: } & \theta \in \mathit{Var} \rightharpoonup \mathit{Value} \\
\text{Values: } & v ::= () \mid (x.e, \theta) \mid \mathbf{inl}(v) \mid \mathbf{inr}(v) \mid (v_1, v_2) \mid \ell \\
& \mid \mathbf{Labeled}\ \ell\ v \mid (t, \theta) \mid n_\ell \\
\text{Expressions: } & e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid () \mid \ell \mid (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \\
& \mid \mathbf{inl}(e_1) \mid \mathbf{inr}(e_2) \mid \mathbf{case}(e, x.e_1, x.e_2) \mid e_1 \sqsubseteq^? e_2 \mid t \\
\text{Thunks: } & t ::= \mathbf{return}(e) \mid \mathbf{bind}(e, x.e) \mid \mathbf{unlabel}(e) \\
& \mid \mathbf{toLabeled}(e) \mid \mathbf{labelOf}(e) \mid \mathbf{getLabel} \mid \mathbf{taint}(e) \\
& \mid \mathbf{new}(e) \mid !\, e \mid e_1 := e_2 \mid \mathbf{labelOfRef}(e) \\
\text{Type System: } & \Gamma \vdash e : \tau \\
\text{Configurations: } & c ::= \langle \Sigma, pc, e \rangle \\
\text{Stores: } & \Sigma \in (\ell : \mathit{Label}) \to \mathit{Memory}\ \ell \\
\text{Memory } \ell\text{: } & M ::= [\,] \mid v : M
\end{aligned}
$$

**Figure 3.1:** Syntax of $\lambda^{dCG}$.

**Syntax.** Figure 3.1 shows the syntax of $\lambda^{dCG}$, a standard simply-typed $\lambda$-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 $\lambda^{dFG}$. The $\lambda^{dCG}$-calculus features both standard (unlabeled) values and *explicitly labeled* values. For example, **Labeled** $H$ **true** represents a secret boolean value of type **Labeled bool**.[1] 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 **return**$(\cdot)$ and **bind**$(\cdot)$ constructs, the

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

$$\text{(Thunk)}\quad t \Downarrow^{\theta} (t,\theta) \qquad \text{(Fun)}\quad \lambda x.e \Downarrow^{\theta} (x.e,\theta) \qquad \text{(Var)}\quad x \Downarrow^{\theta} \theta(x)$$

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

$$\text{(Case}_1\text{)}\quad \frac{e_1 \Downarrow^{\theta} \mathbf{inl}(v_1) \qquad e_1 \Downarrow^{\theta[x\mapsto v_1]} v}{\mathbf{case}(e, x.e_1, x.e_2) \Downarrow^{\theta} v} \qquad \text{(Case}_2\text{)}\quad \frac{e_1 \Downarrow^{\theta} \mathbf{inr}(v_2) \qquad e_2 \Downarrow^{\theta[x\mapsto v_2]} v}{\mathbf{case}(e, x.e_1, x.e_2) \Downarrow^{\theta} v}$$

**Figure 3.2:** Pure semantics: $e \Downarrow^{\theta} v$ (selected rules).

monad provides primitives that regulate the creation and the inspection of labeled values, i.e., **toLabeled**$(\cdot)$, **unlabel**$(\cdot)$ and **labelOf**$(\cdot)$, and the interaction with the labeled store, allowing the creation, reading and writing of labeled references $n_\ell$ through the constructs **new**$(e)$, $!e$, $e_1 := e_2$, respectively.[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., $(t,\theta)$, provides a way to suspend nested computations.

## 3.1 Dynamics

In order to track information flows dynamically at coarse granularity, $\lambda^{dCG}$ employs a technique called *floating-label*, which was originally developed for IFC operating systems (e.g., Zeldovich *et al.*, 2006; Zeldovich *et al.*, 2008) and that was later applied in a language-based setting. In this technique, throughout a program's execution, the program counter label *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, $\lambda^{dCG}$ stores

[2] We extend $\lambda^{dCG}$ with *flow-sensitive* references and *heaps* in Section 3.3.

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

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

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

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

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

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

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

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

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

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

**Figure 3.3:** Big-step semantics for $\lambda^{dCG}$.

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 said to be *flow-sensitive*, i.e., it may assume different values in the final configuration depending on the control flow of the program.[3]

Like $\lambda^{dFG}$, the operational semantics of $\lambda^{dCG}$ consists of a security monitor that fully evaluates secure programs but prevents the execution of insecure programs and similarly enforces *termination-insensitive* noninterference (Theorem 3). The big-step operational semantics of $\lambda^{dCG}$ is structured in two parts: (i) a straightforward call-by-value side-effect-free semantics for pure expressions (Figure 3.2), and (ii) a top-level security monitor for monadic programs (Figure 3.3). The semantics of the security monitor is further split into two mutually recursive reduction relations, one for arbitrary expressions (Figure 3.3a) and one specific to thunks (Figure 3.3b). These constitute the *forcing* semantics of the monad, which reduces a thunk to a pure value and perform side-effects. In particular, given the initial store $\Sigma$, program counter label $pc$, expression $e$ of type **LIO** $\tau$ for some type $\tau$ and input values $\theta$ (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 $\Sigma'$, updated program counter $pc'$ and a final value $v$ of type $\tau$, which also might not be labeled. The execution starts with rule [FORCE], which reduces the pure expression to a thunk closure, i.e., $(t, \theta')$ and then forces the thunk $t$ in its environment $\theta'$ with the thunk semantics.

The pure semantics is fairly standard—we report some selected rules in Figure 3.2 for comparison with $\lambda^{dFG}$. A pure reduction, written $e \Downarrow^{\theta} v$, evaluates an expression $e$ with an appropriate environment $\theta$ to a pure value $v$. Notice that, unlike $\lambda^{dFG}$, all reduction rules of the pure semantics ignore security, even those that affect the control flow of the program, like rules [APP], [CASE$_1$], and [CASE$_2$]: they do not feature the program counter label or label annotations. This is because these reductions are *pure*—they cannot perform side-effects and so leak sensitive data—and thus are inherently secure and need

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

not to be monitored (Vassena *et al.*, 2017).[4] For example, since pure programs do not have access to the store, they cannot leak through *implicit flows*, which are then a concern only for monadic programs.

How does the semantics prevent pure programs from performing side-effects, without getting the program stuck and raising a false alarm? 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 $\theta$ (see rule [Thunk]).[5] Notice that *thunk closures* and *function closures* are distinct values created by different rules, [Thunk] and [Fun] respectively.[6] 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 $\ell$ 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 been 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

[4]The strict separation between side-effect-free and side-effectful code is a distinctive feature of coarse-grained IFC, which is crucial to lightweight approaches to enforce security via software libraries (Russo *et al.*, 2009; Buiras *et al.*, 2015; Russo, 2015; Stefan *et al.*, 2012)

[5]Notice that *type preservation* for the pure semantics preserves types *exactly* i.e., if $\Gamma \vdash e : \tau$, $e \Downarrow^{\theta} v$ and $\vdash \theta : \Gamma$, then $\vdash v : \tau$, which reflects the *suspending* behavior for the monadic type **LIO** $\tau$. In contrast, type preservation for the *thunk* and *forcing* semantics assumes that the expression (resp. thunk) has a monadic type, i.e., **LIO** $\tau$ for some type $\tau$, and guarantees that the final value has type $\tau$.

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

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** $pc'$ $v$.[7] 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**$(\cdot)$ 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 $\lambda^{dFG}$). Note that, in $\lambda^{dCG}$, **taint**$(\cdot)$ takes *only* the label with which the program counter must be tainted whereas, in $\lambda^{dFG}$, 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 $\lambda^{dCG}$.

### 3.1.1 References

$\lambda^{dCG}$ features *flow-insensitive* labeled references similar to $\lambda^{dFG}$ and allows programs to create, read, update and inspect the label inside the **LIO** monad (see Figure 3.4). 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

[7] Stefan *et al.* (2017) have proposed an alternative flow-insensitive primitive, i.e., **toLabeled**$(\ell, e)$, which labels the result with the user-assigned label $\ell$. In $\lambda^{dCG}$, we use primitive **toLabeled**$(e)$ because its flow-sensitive labeling semantics aligns exactly with the semantics of $\lambda^{dFG}$, thus simplifying the translation between the languages.

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

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

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

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

**Figure 3.4:** Big-step semantics for $\lambda^{dCG}$ (references).

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 $\ell$-labeled memory and returns it. Since this brings data at security level $\ell$ in scope, the program counter is tainted accordingly, i.e., $pc \sqcup \ell$. Rule [Write] performs security checks analogous to those in $\lambda^{dFG}$ 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 $\lambda^{dCG}$ satisfy Property 6 (*"the final value of the program counter label of any $\lambda^{dCG}$ program is always at least as sensitive as the initial value"*).

**Property 6.**

- 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. □

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

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

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

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

$$\frac{\Sigma_1 \approx_L \Sigma_2 \qquad pc \sqsubseteq L \qquad v_1 \approx_L v_2}{\langle \Sigma_1, pc, v_1 \rangle \approx_L \langle \Sigma_2, pc, v_2 \rangle}\ (\textsc{Pc}_L)$$

**Figure 3.5:** $L$-equivalence for $\lambda^{dCG}$ values (selected rules) and configurations.

## 3.2 Security

We now prove that $\lambda^{dCG}$ 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 $\lambda^{dFG}$— when secret inputs are changed, terminating programs do not produce any publicly observable effect—and based on a similar indistinguishability relation.

### 3.2.1 $L$-Equivalence

Figure 3.5 presents the definition of $L$-equivalence for the interesting cases only. Firstly, $L$-equivalence for $\lambda^{dCG}$ labeled values relates public and secret values analogously to $\lambda^{dFG}$ values. Specifically, rule

[Labeled$_L$] relates public labeled values that share the *same* observable label, i.e., $\ell \sqsubseteq L$, 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 $L$, i.e., $\ell_1 \not\sqsubseteq L$ 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 $v_1 \approx_L v_1'$ through rule [Inl] and $\mathbf{inr}(v_2) \approx_L \mathbf{inr}(v_2')$ iff $v_2 \approx_L v_2'$ through rule [Inr], i.e., we do not provide any rule to relate values with different injections, just like in $\lambda^{dFG}$. Function and thunk closures are related by rules [F-Closure] and [T-Closure], respectively. In the rules the function and the monadic computations are $\alpha$-equivalent and their environments are related, i.e., $\theta_1 \approx_L \theta_2$ iff $dom(\theta_1) \equiv dom(\theta_2)$ and $\forall x.\theta_1(x) \approx_L \theta_2(x)$. Labeled references, memories and stores are related by $L$-equivalence analogously to $\lambda^{dFG}$. Lastly, $L$-equivalence relates *initial* configurations with related stores, equal program counters and $\alpha$-equivalent expressions (resp. thunks), 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 $e_1 \equiv_\alpha e_2$ (resp. $t_1 \equiv_\alpha t_2$ for thunks $t_1$ and $t_2$), and *final* configurations with related stores and (i) equal public program counter label, i.e., $pc \sqsubseteq L$, and related values through rule [Pc$_L$], or (ii) arbitrary secret program counter labels, i.e., $pc_1 \not\sqsubseteq L$ and $pc_2 \not\sqsubseteq L$, and arbitrary values through rule [Pc$_H$].

$L$-equivalence for $\lambda^{dCG}$ is *reflexive*, *symmetric*, and *transitive*, similarly to $\lambda^{dFG}$ (Property 2), and so these properties can be combine to derive a *Square Commutative Diagram for* $\lambda^{dCG}$ *Stores* as well, analogous to Figure 2.6.

### 3.2.2 Termination-Insensitive Non-Interference

We now formally prove that also the security monitor of $\lambda^{dCG}$ is secure, i.e., it enforces *termination-insensitive non-interference* (TINI). The proof technique is the same used for $\lambda^{dFG}$ and similarly based on *store confinement* and *$L$-equivalence preservation*. However, since the semantics of the security monitor of $\lambda^{dCG}$ is defined by two mutually recursive relations, i.e., the forcing and the thunk semantics, each lemma is stated as a pair of lemmas (one for each semantics relation), which are then proved by mutual induction.

**Store Confinement.** Store confinement ensures that programs running in a secret context cannot leak data implicitly through the labeled store. To prevent these leaks, the security monitor of $\lambda^{dCG}$ aborts programs that attempt to write public memories in secret contexts, as these changes may depend on secret data and would be observable by the attacker. Rules [New] and [Write] enforce exactly this security policy by allowing programs to write memories only if they are labeled above the program counter label.

**Lemma 3.1** (Store Confinement)**.** For all program counter labels $pc \not\sqsubseteq L$, initial configurations $c$, and final configurations $c' = \langle \Sigma', pc', v \rangle$:

- If $c = \langle \Sigma, pc, e \rangle$ and $c \Downarrow^{\theta} c'$, then $\Sigma \approx_L \Sigma'$.
- If $c = \langle \Sigma, pc, t \rangle$ and $c \Downarrow^{\theta} c'$, then $\Sigma \approx_L \Sigma'$.

*Proof.* By mutual induction, using *reflexivity* in most base cases and *transitivity* in case [Bind], where the program counter label remains above the attacker's level in the intermediate configuration by Property 6. In the base cases [New] and [Write], the programs run with program counter label secret, i.e., $pc \not\sqsubseteq L$, and write a secret memory labeled $\ell$ above $pc$, i.e., $pc \sqsubseteq \ell$, and thus also not observable by the attacker, i.e., $\ell \not\sqsubseteq L$. □

**$L$-Equivalence Preservation.** Next, we prove $L$-equivalence preservation in secret contexts, i.e., programs running with program counter label secret ($pc \not\sqsubseteq L$) cannot leak secret data implicitly through their final value or observable changes to the store. In $\lambda^{dCG}$, values are not intrinsically labeled like $\lambda^{dFG}$ values, therefore the final values computed by these programs may not be explicitly labeled. If these values are unlabeled, how can we establish if they depend on secret data, or if they are $L$-equivalent? Luckily, we can safely approximate the sensitivity of these values with the program counter label.

The program counter label always represents an upper bound over the sensitivity of all data not explicitly labeled in a program, including its final value. This is precisely why $\lambda^{dCG}$ stores it in the program

configuration, so that the **LIO** monad can control it through the floating-label mechanism explained above. Therefore, the fact that the program counter label is secret in a final configuration indicates that the program has unlabeled secret data and thus the final value *may* depend on secrets. This approximation is sound, but also very *conservative*—in a secret context, the result of a program must always be considered secret even if the program has not actually used any secret data to compute it—and central to the design of the *coarse-grained* IFC approach embodied by $\lambda^{dCG}$.[8]

The $L$-equivalence relation for final configurations reflects this reading of the program counter label for unlabeled values. Specifically, if both program counter labels are secret, then rule [PC$_H$] (Figure 3.5) simply ignores the values in the configurations (because they *may* depend on secret data) and only requires the stores to be $L$-equivalent.[9] Then, to prove $L$-equivalence preservation in secret contexts, we simply observe that the program counter label can only increase during program execution (Property 6), thus programs that start in secret contexts must also necessarily terminate in secret contexts. Therefore, we can apply rule [PC$_H$] provided that the final stores are $L$-equivalent, i.e., if we can show that programs cannot leak implicitly through the public memories of the store. To do that, we follow the same proof technique used for $\lambda^{dFG}$, i.e., we first prove that the final stores are $L$-equivalent to the initial stores by applying *store confinement* (Lemma 3.1) to each execution and then derive $L$-equivalence of the final stores by constructing a *square commutative diagram* (Figure 2.6).

**Lemma 3.2** ($L$-Equivalence Preservation in Secret Contexts)**.** For all program counter labels $pc_1 \not\sqsubseteq L$ and $pc_2 \not\sqsubseteq L$, and stores $\Sigma_1 \approx_L \Sigma_2$:

- If $\langle \Sigma_1, pc_1, e_1 \rangle \Downarrow^{\theta_1} c_1$ and $\langle \Sigma_2, pc_2, e_2 \rangle \Downarrow^{\theta_2} c_2$, then $c_1 \approx_L c_2$.
- If $\langle \Sigma_1, pc_1, t_1 \rangle \Downarrow^{\theta_1} c_1$ and $\langle \Sigma_2, pc_2, t_2 \rangle \Downarrow^{\theta_2} c_2$, then $c_1 \approx_L c_2$.

[8] Due to this conservative behavior, coarse-grained IFC languages like $\lambda^{dCG}$ may seem inherently limited, compared to fine-grained languages like $\lambda^{dFG}$, which track data-dependencies more precisely, i.e., the result gets labeled secret *only if* secret data has been used to compute its value. In Section 5, we prove that $\lambda^{dCG}$ can track data-dependencies as precisely as $\lambda^{dFG}$.

[9] In particular, rule [PC$_H$] accepts arbitrary final values just like rule [LABELED$_H$] for explicitly labeled values.

*Proof.* Assume $pc_1 \not\sqsubseteq L$, $pc_2 \not\sqsubseteq L$, $\Sigma_1 \approx_L \Sigma_2$, and let $c_1 = \langle \Sigma_1', pc_1', v_1 \rangle$ and $c_2 = \langle \Sigma_2', pc_2', v_2 \rangle$. First, we apply *store confinement* (Lemma 3.1) to each big-step reduction and obtain $\Sigma_1 \approx_L \Sigma_1'$ and $\Sigma_2 \approx_L \Sigma_2'$. These are then combined with the assumption $\Sigma_1 \approx_L \Sigma_2$ to form the commutative square diagram for $\lambda^{dCG}$ stores (Figure 2.6), which gives $\Sigma_1' \approx_L \Sigma_2'$. Then, we observe that the program counter labels in the final configurations are *secret*, i.e., $pc_1' \not\sqsubseteq L$ and $pc_2' \not\sqsubseteq L$ by Property 6, and therefore we have $c_1 \approx_L c_2$ by rule [$\textsc{Pc}_H$]. □

Now, we consider $L$-equivalence preservation in public contexts. Since $\lambda^{dCG}$ separates *pure computations* from *monadic computations* in different semantics judgments, we first need to prove that $L$-equivalence is preserved also by the pure semantics. Notice that this separation is not only beneficial for security, as explained above, but simplifies the security analysis as well. For example, pure computations cannot inspect secret data in a public context, as this operation requires performing a side-effect. This is because, in public contexts, secret data is *explicitly* protected through the **Labeled** type, which prevents programs from inspecting secrets directly as they must first *extract* them via **unlabel**$(\cdot)$: **Labeled** $\tau \to$ **LIO** $\tau$. The monadic type of this thunk indicates that this computation may perform side-effects (i.e., tainting the program counter label), which crucially, can be performed only by the security monitor, in the thunk semantics.[10] As a result, the execution of pure programs in public contexts cannot depend on secret data—they simply cannot access secrets in the first place—and so their (unlabeled) results are always indistinguishable by the attacker.

**Lemma 3.3** (Pure $L$-Equivalence Preservation)**.** For all expressions $e_1 \equiv_\alpha e_2$ and environments $\theta_1 \approx_L \theta_2$, if $e_1 \Downarrow^{\theta_1} v_1$ and $e_2 \Downarrow^{\theta_2} v_2$, then $v_1 \approx_L v_2$.

*Proof.* By induction on the big-step reductions, which *must* always step according to the same rule. In the spurious cases, e.g., when expression **case**$(e, x.e_1, x.e_2)$ steps through rules [$\textsc{Case}_1$] and [$\textsc{Case}_2$], we show a contradiction. In particular, assume that the executions follow different paths, e.g., we have reductions $e \Downarrow^{\theta_1}$ **inl**$(v_1)$ and $e \Downarrow^{\theta_2}$ **inr**$(v_2)$ for the

[10] The pure semantics simply *suspends* the evaluation of thunks via rule [$\textsc{Thunk}$].

scrutinee $e$. Then, we apply the induction hypothesis to these reductions and obtain a proof for $\mathbf{inl}(v_1) \approx_L \mathbf{inr}(v_2)$. But this is impossible: $L$-equivalence relates values of sum types only if they have the same injection, i.e., rules [Inl] and [Inr] in Figure 3.5. Therefore, the two executions must evaluate the scrutinee to the same injection and follow the same path. □

Similarly, the fact that $\lambda^{dCG}$ requires programs to explicitly unlabel secret data simplifies also the analysis of *implicit flows* in monadic computations. This is because the control flow of $\lambda^{dCG}$ programs can only depend on unlabeled data, whose sensitivity is *coarsely* approximated by the program counter label, as explained above. Since the program counter label only gets tainted in response to specific monadic actions (e.g., $\mathbf{unlabel}(e)$), and not by regular control-flow construct (e.g., $\mathbf{case}(e, x.e_1, x.e_2)$), the evaluation of pure expressions *cannot* cause implicit information flows. In particular, by encapsulating secret data in the **Labeled** data type, $\lambda^{dCG}$ makes all data dependencies—even those implicit in the program control flow—*explicit* through the program counter label.

**Lemma 3.4** ($L$-Equivalence Preservation in Public Contexts)**.** For all public program counter labels $pc \sqsubseteq L$, environments $\theta_1 \approx_L \theta_2$, and stores $\Sigma_1 \approx_L \Sigma_2$:

- If $e_1 \equiv_\alpha e_2$, $\langle \Sigma_1, pc, e_1 \rangle \Downarrow^{\theta_1} c_1$, $\langle \Sigma_2, pc, e_2 \rangle \Downarrow^{\theta_2} c_2$, then $c_1 \approx_L c_2$;
- If $t_1 \equiv_\alpha t_2$, $\langle \Sigma_1, pc, t_1 \rangle \Downarrow^{\theta_1} c_1$, $\langle \Sigma_2, pc, t_2 \rangle \Downarrow^{\theta_2} c_2$, then $c_1 \approx_L c_2$.

*Proof.* The two lemmas are proved mutually, by simultaneous induction on the big-step reductions. The *forcing semantics* has only a single rule, i.e., [Force] in Figure 3.3a. Therefore, for the first lemma, we simply apply *Pure $L$-Equivalence Preservation* (Lemma 3.3) to the pure reductions $e_1 \Downarrow^{\theta_1} (t_1, \theta_1')$ and $e_2 \Downarrow^{\theta_2} (t_2, \theta_2')$, which gives $(t_1, \theta_1') \approx_L (t_2, \theta_2')$, i.e., $t_1 \equiv_\alpha t_2$ and $\theta_1' \approx_L \theta_2'$, so we complete the proof by mutual induction with the lemma for thunks. For the second lemma, we observe that the thunks are $\alpha$-equivalent, i.e., $t_1 \equiv_\alpha t_2$, therefore their reductions always step according to the same rule. For example, in case [Unlabel], the labeled values that get unlabeled are $L$-equivalent,

i.e., **Labeled** $\ell_1\ v_1 \approx_L$ **Labeled** $\ell_2\ v_2$, by Lemma 3.3 applied to the pure reductions, and then we proceed by cases on the $L$-equivalence judgment. (Notice that the store does not change in rule [Unlabel], so the final stores are $L$-equivalent by assumption). In case [Labeled$_L$], the two values are labeled public, i.e., $\ell_1 = \ell_2 \sqsubseteq L$, and the values are related, i.e., $v_1 \approx_L v_2$. Therefore, the program counter label in the final configurations remain observable by the attacker, i.e., $pc \sqcup \ell_1 \sqsubseteq L$ since $pc \sqsubseteq L$ and $\ell_1 \sqsubseteq L$, and the final configurations are $L$-equivalent by rule [Pc$_L$]. In case [Labeled$_H$], the two values are labeled secret, i.e., $\ell_1 \not\sqsubseteq L$ and $\ell_2 \not\sqsubseteq L$, and the program counter label in the final configurations are secret, i.e., $pc \sqcup \ell_1 \not\sqsubseteq L$ and $pc \sqcup \ell_2 \not\sqsubseteq L$, and so the configurations are $L$-equivalent by rule [Pc$_H$]. □

Finally, we prove termination-insensitive non-interference for $\lambda^{dCG}$ by combining the $L$-equivalence preservation lemmas above.

**Theorem 3** ($\lambda^{dCG}$-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 $c_1' \approx_L c_2'$.

### 3.3 Flow-Sensitive References

We now continue our exploration of coarse-grained IFC by adding flow-sensitive references to $\lambda^{dCG}$ and then showing that the extended language is secure. This extension is in many ways analogous to the extension presented in Section 2.3 for $\lambda^{dFG}$. Therefore, we focus mainly on the (apparently different) security checks performed by the security monitor of $\lambda^{dCG}$ and how they relate to those performed by the monitor of $\lambda^{dFG}$. In Section 3.3.3, we formally establish the security guarantees of $\lambda^{dCG}$ extended with flow-sensitive references. Though the security analysis of $\lambda^{dCG}$ is also complicated by bijections like $\lambda^{dFG}$, we find that the separation between pure and monadic computations in $\lambda^{dCG}$ limits the extra complexity of the analysis only to the monadic fragment of the semantics. In particular, this extension does not affect the pure fragment of the semantics and so adapting this part of the analysis is straightforward. Finally, we leverage our mechanized proof scripts to compare and analyze the security proofs of $\lambda^{dFG}$ and $\lambda^{dCG}$ and discuss our findings in Section 4.

### 3.3.1 Syntax

Figure 3.6 introduces the new syntactic constructs and semantics rules for flow-sensitive references. First, we annotate the reference type constructor with a tag $s \in \{I, S\}$, which indicates the sensitivity of a reference term. A plain address $n : \mathbf{Ref}\ S\ \tau$ represents a flow-sensitive reference pointing to a value of type $\tau$, stored in the $n$-th cell of the heap. Notice that these references are *not* annotated with a label representing the sensitivity of their content, like flow-insensitive references. This is because flow-sensitive references are, by design, allowed to store data at different security levels, i.e., the label of these references can *change* throughout program execution. Thus, the references themselves are *unlabeled* and, instead, the security monitor *explicitly* labels their content in the heap when they are created and updated (see below).[11] To ensure that values in the heap are always labeled, heaps $\mu$ are syntactically defined as lists of explicitly labeled values $\mathbf{Labeled}\ \ell\ v$, whose label $\ell$ represents the sensitivity of value $v$ and, at the same time, the label of any reference pointing to it. The heap is stored in program configurations $\langle \Sigma, \mu, pc, e \rangle$, where programs can access it through the same thunk constructs used for flow-insensitive references, i.e., $\mathbf{new}(e)$, $e_1 := e_2$, $!e$, and $\mathbf{labelOfRef}(e)$. As we explain next, these operations must be regulated by the security monitor of $\lambda^{dCG}$ to enforce security.

### 3.3.2 Dynamics

Figure 3.6 extends the thunk semantics with new rules that allow programs to access the heap through flow-sensitive references without leaking data. Rule [New-FS] creates a new reference by allocating a new cell in the heap at fresh address $n = |\mu|$, which is initialized with the given labeled value argument, i.e., $\mu[n \mapsto \mathbf{Labeled}\ \ell\ v]$. Importantly, the rule requires the label of the value to be above the program counter label, i.e., $pc \sqsubseteq \ell$, to avoid leaks.[12] Without this constraint, a program

[11] In contrast, flow-insensitive references are annotated with a *fixed* label, e.g., $\ell$ for $n_\ell : \mathbf{Ref}\ I\ \tau$, which represents (an upper bound over) the sensitivity of its content. Since this label does not change, the content of the reference can be stored directly in the memory labeled $\ell$, without being explicitly labeled.

[12] This restriction, known as *no write-down* (Bell and La Padula, 1976), is a core design principles of several static IFC libraries (Russo, 2015; Vassena *et al.*, 2017),

$$\begin{aligned}
\text{Types: } \tau &::= \cdots \mid \mathbf{Ref}\ s\ \tau \\
\text{Values: } v &::= \cdots \mid n \\
\text{Heap: } \mu &::= [\,] \mid \mathbf{Labeled}\ \ell\ v : \mu \\
\text{Configuration: } c &::= \langle \Sigma, \mu, pc, e \rangle
\end{aligned}$$

**(a)** Syntax.

(New-FS)

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

(Read-FS)

$$\frac{e \Downarrow^{\theta} n \qquad \mu[n] = \mathbf{Labeled}\ \ell\ v}{\langle \Sigma, \mu, pc, !e \rangle \Downarrow^{\theta} \langle \Sigma, \mu, pc \sqcup \ell, v \rangle}$$

(LabelOfRef-FS)

$$\frac{e \Downarrow^{\theta} n \qquad \mu[n] = \mathbf{Labeled}\ \ell\ \_}{\langle \Sigma, \mu, pc, \mathbf{labelOfRef}(e) \rangle \Downarrow^{\theta} \langle \Sigma, \mu, pc \sqcup \ell, \ell \rangle}$$

(Write-FS)

$$\frac{e_1 \Downarrow^{\theta} n \qquad e_2 \Downarrow^{\theta} \mathbf{Labeled}\ \ell'\ v \qquad \mu[n] = \mathbf{Labeled}\ \ell\ \_ \qquad pc \sqsubseteq \ell \qquad \mu' = \mu[n \mapsto \mathbf{Labeled}\ (pc \sqcup \ell')\ v]}{\langle \Sigma, \mu, pc, e_1 := e_2 \rangle \Downarrow^{\theta} \langle \Sigma, \mu', pc, () \rangle}$$

**(b)** Dynamics. The shaded constraint corresponds to the *no-sensitive upgrade* security check in $\lambda^{dFG}$.

**Figure 3.6:** $\lambda^{dCG}$ extended with flow-sensitive references.

could create a public reference in a secret context, which constitutes a leak. Compare this rule with the corresponding rule for $\lambda^{dFG}$, i.e., rule [New-FS] in Figure 2.8b. The security monitor of $\lambda^{dFG}$ does not check that the label of the value is above the program counter label—in fact the rule does not contain any security check at all! Why must $\lambda^{dCG}$ include that check and instead $\lambda^{dFG}$ can skip it? This is because that check is *redundant* in $\lambda^{dFG}$: the constraint $pc \sqsubseteq \ell$ performed by the security monitor of $\lambda^{dCG}$ is an invariant (Property 1) of the semantics of $\lambda^{dFG}$. Intuitively, $\lambda^{dFG}$ programs cannot create public references in secret contexts because the result of a computation is always labeled above the current program counter label. Since $\lambda^{dCG}$ does not enjoy this invariant, i.e., computations can return values labeled public even in secret contexts, that check *must* be included in all the rules that perform write side-effects (i.e., also in rules [New], [Write] for flow-insensitive references in Figure 3.4) to avoid leaks.

Rule [Read-FS] reads a flow-sensitive reference by retrieving its content from the heap, i.e., $\mu[n] = \textbf{Labeled}\ \ell\ v$, and returning its value $v$. Importantly, the rule taints the program counter label, i.e., $pc \sqcup \ell$, to indicate that data at security level $\ell$ is now in scope. Rule [LabelOfRef-FS] is similar, but returns the label of the reference, i.e., the same label $\ell$ that also annotates its content, and so it taints the program counter label with the label itself, i.e., $pc \sqcup \ell$.

Lastly, rule [Write-FS] updates the content of a flow-sensitive reference $n$ with a new value $\textbf{Labeled}\ \ell'\ v$, which replaces the current value stored in the heap, i.e., $\mu[n] = \textbf{Labeled}\ \ell\ \_$. Notice that the new value can be less or *even more* sensitive than the old value, i.e., labels $\ell$ and $\ell'$ can be in any relation. This is exactly why flow-sensitive references are, in general, more permissive than flow-insensitive references also in $\lambda^{dCG}$.[13]

In the rule, the constraint $pc \sqsubseteq \ell$ is analogous to the NSU check for $\lambda^{dFG}$. (For convenience, these equivalent checks are shaded in Figure 2.8b and 3.6b). Intuitively, this check allows a reference update only if

which was not identified as such in dynamic IFC libraries (Stefan *et al.*, 2011; Stefan *et al.*, 2017).

[13] In particular, Example 2.1 can be adapted to $\lambda^{dCG}$ as well, i.e., program $r \leftarrow \textbf{new}(p); r := s; !r$ is aborted with flow-insensitive references, but accepted with flow-sensitive references.

the decision to update *that* reference depends on data less sensitive than the reference itself.[14] Though the NSU checks in $\lambda^{dFG}$ and $\lambda^{dCG}$ protect against the same type of implicit data leaks, they involve different labels, i.e., the intrinsic label of the reference in $\lambda^{dFG}$ and the program counter label in $\lambda^{dCG}$. In $\lambda^{dFG}$, the sensitivity of the reference is represented by its intrinsic label, which can then be used directly in the NSU check. In contrast, $\lambda^{dCG}$ does not feature intrinsic labels, therefore, we use the program counter label as a coarse, but *sound*, approximation of the sensitivity of the reference, hence the NSU check $pc \sqsubseteq \ell$. Notice that when the new value is written in the heap, the rule taints its label with the program counter label, i.e., $\mu[n \mapsto \textbf{Labeled}\ (pc \sqcup \ell')\ v]$, for the same reason. Intuitively, the sensitivity of the content is determined by the new value, which is explicitly labeled $\ell'$, and by the sensitivity of the (unlabeled) reference, approximated by the program counter label $pc$, i.e., the new content has sensitivity at most $pc \sqcup \ell'$.[15]

### 3.3.3 Security

The security analysis of $\lambda^{dCG}$ extended with flow-sensitive references is very much similar to the analysis presented in Section 2.3.3 for $\lambda^{dFG}$. In particular, the security monitor of $\lambda^{dCG}$ allocates both public and secret data in the same linear heap and so secret-dependent allocations can influence the addresses of subsequent public references, just like in $\lambda^{dFG}$. Although this can create a dependency between concrete heap addresses and secret data, security is not at stake because references are *opaque* in $\lambda^{dCG}$ as well. To formally show that, we need to adapt the $L$-equivalence relation, and consequently the security analysis, to use a *bijection* to reason about corresponding references with secret-dependent, yet indistinguishable, heap addresses.

**$L$-Equivalence up to Bijection.** Formally, we redefine $L$-equivalence as a relation $\approx_L^\beta \subseteq \mathit{Value} \times \mathit{Value}$ and add the bijection $\beta$ to $\approx_L$ in all the rules previously defined for $\lambda^{dCG}$. These rules do not use the bijection,

[14] $\lambda^{dCG}$ would be insecure without this constraint. In particular, programs could leak data through secret-dependent reference updates, similarly to Example 2.2.

[15] In addition, by tainting the label $\ell'$ with the program counter label $pc$, the rule automatically respects the *no write-down* restriction, i.e., $pc \sqsubseteq pc \sqcup \ell'$, hence no explicit security check is needed like in rules [New], [Write], and [New-FS].

which is needed instead only in the rules for flow-sensitive references and heaps. In particular, two flow-sensitive references are indistinguishable only if their heap addresses are matched by the bijection, i.e., we add the following new rule:

$$\frac{(n_1, n_2) \in \beta}{n_1 \approx_L^\beta n_2} \text{(Ref-FS)}$$

Similarly, in the definition of $L$-equivalence for heaps, the addresses related by the bijection identify corresponding heap cells, which must be recursively related, as they can be read through related references.[16]

**Definition 3** (Heap $L$-equivalence)**.** Two heaps $\mu_1$ and $\mu_2$ are $L$-equivalent up to bijection $\beta$, written $\mu_1 \approx_L^\beta \mu_2$, if and only if:

1. $dom(\beta) \subseteq \{0, \ldots, |\mu_1| - 1\}$,
2. $rng(\beta) \subseteq \{0, \ldots, |\mu_2| - 1\}$, and
3. For all addresses $n_1$ and $n_2$, if $(n_1, n_2) \in \beta$, then $\mu_1[n_1] \approx_L^\beta \mu_2[n_2]$.

In the definition above, only the rules for labeled values can relate heap cells $\mu_1[n_1] \approx_L^\beta \mu_2[n_2]$. This is because heaps are syntactically defined as a list of explicitly labeled values in Figure 3.6a, i.e., $\mu_1[n_1] = \textbf{Labeled}\ \ell_1\ v_1$ for some label $\ell_1$ and value $v_1$ and similarly $\mu_2[n_2] = \textbf{Labeled}\ \ell_2\ v_2$, and the only rules that give $\textbf{Labeled}\ \ell_1\ v_1 \approx_L^\beta \textbf{Labeled}\ \ell_2\ v_2$ are [Labeled$_L$] and [Labeled$_H$] in Figure 3.5.

Similarly to $\lambda^{dFG}$, the $L$-equivalence relation up to bijection for $\lambda^{dCG}$ satisfies *restricted reflexivity*, *symmetricity*, *transitivity*, and *weakening* (i.e., Property 5), and so we can construct square commutative diagrams for stores and heaps (i.e., Lemma 2.7) in $\lambda^{dCG}$ as well. Notice that reflexivity is restricted only to *valid* program constructs, i.e., values, environments, memories, stores, heaps, and configurations that are *free* of dangling flow-sensitive references, just like it is in $\lambda^{dFG}$ and for the same technical reasons. These side conditions are formalized

[16] Notice that these addresses are valid in the heap thanks to the first two side-conditions, which ensure that the domain and range of the bijection are compatible with the address space of the heaps. See also the explanation of Definition 2.

by straightforward judgments, e.g., $n \vdash \mathbf{Valid}(\theta)$ and $n \vdash \mathbf{Valid}(v)$ indicating that the references contained in value $v$ and environment $\theta$ are valid in a heap of size $n$, which are mutually and recursively defined like in Figure 2.9. In these judgments, the size parameter $n$ is instantiated by the top-level judgment for program inputs and outputs with the size of the current heap $\mu$, i.e., $n = |\mu|$ as shown in Figure 3.7. The fact that the security lemmas of $\lambda^{dCG}$ assume valid configurations does not actually weaken the security guarantees of the language: these judgments are naturally preserved by the pure and monadic semantics of $\lambda^{dCG}$.

(Valid-Inputs)

$$\frac{c = \langle \Sigma, \mu, pc\ e \rangle \qquad n = |\mu| \qquad n \vdash \mathbf{Valid}(\Sigma) \qquad n \vdash \mathbf{Valid}(\mu) \qquad n \vdash \mathbf{Valid}(\theta)}{\vdash \mathbf{Valid}(c, \theta)}$$

(Valid-Outputs)

$$\frac{c = \langle \Sigma, \mu, pc, v \rangle \qquad n = |\mu| \qquad n \vdash \mathbf{Valid}(\Sigma) \qquad n \vdash \mathbf{Valid}(\mu) \qquad n \vdash \mathbf{Valid}(v)}{\vdash \mathbf{Valid}(c)}$$

**Figure 3.7:** Judgments for valid program inputs and outputs in $\lambda^{dCG}$.

**Property 7** (Valid Invariant)**.**

1. If $e \Downarrow^{\theta} v$ and $n \vdash \mathbf{Valid}(\theta)$, then $n \vdash \mathbf{Valid}(v)$.
2. If $c \Downarrow^{\theta} c'$ and $\vdash \mathbf{Valid}(c, \theta)$, then $\vdash \mathbf{Valid}(c')$.

**Termination-Insensitive Non-Interference.** We now prove that $\lambda^{dCG}$ extended with flow-sensitive references enforces termination-insensitive non-interference. This result is also based on two lemmas, i.e., *store and heap confinement* and *$L$-equivalence preservation*, which are complicated by the fact that $L$-equivalence is defined up to a bijection, similarly to $\lambda^{dFG}$. These lemmas rely on bijections to relate flow-sensitive references allocated in public contexts: these references are observable

by the attacker, but their heap addresses may differ due to previous, secret-dependent allocations. In contrast, references allocated in secret contexts cannot be observed by the attacker and must be ignored by the bijection. For example, in the store and heap confinement lemma, we show that initial and final stores and heaps are $L$-equivalent, for programs executed in secret contexts. Which bijection should we use to relate the addresses of flow-sensitive references in this lemma? Intuitively, the addresses of references already allocated in the initial configuration remain unchanged in the final configuration (because references cannot be tempered in $\lambda^{dCG}$), so they can be related by the *identity bijection*. Furthermore, since references allocated in a secret context are not observable by the attacker, the lemma shows $L$-equivalence up to a identity bijection that ignores any new heap allocation performed by the program, i.e., up to the identity bijection *restricted to the domain of the initial heap*.

**Lemma 3.5** (Store and Heap Confinement)**.** For all program counter labels $pc \not\sqsubseteq L$, valid initial configurations and environments $\vdash \mathbf{Valid}(c, \theta)$, and final configurations $c' = \langle \Sigma', \mu', pc', v \rangle$:

- If $c = \langle \Sigma, \mu, pc, e \rangle$ and $c \Downarrow^{\theta} c'$, then $\Sigma \approx_L^{\iota_{|\mu|}} \Sigma'$ and $\mu \approx_L^{\iota_{|\mu|}} \mu'$.
- If $c = \langle \Sigma, \mu, pc, t \rangle$ and $c \Downarrow^{\theta} c'$, then $\Sigma \approx_L^{\iota_{|\mu|}} \Sigma'$ and $\mu \approx_L^{\iota_{|\mu|}} \mu'$.

*Proof.* Analogous to Lemma 2.8. Since $\lambda^{dCG}$ encapsulates side-effects in a monad, we need to explicitly propagate the valid judgment only in very few cases compared to $\lambda^{dFG}$. In particular, we only apply Property 7.1 in rule [Force] and Property 7.2 in rule [Bind]. □

Next, we prove $L$-equivalence preservation in *secret contexts*. Since the programs considered in this lemma run in a secret context, the addresses of the flow-sensitive references allocated during these executions must be ignored by the bijection, i.e., the final configurations must be $L$-equivalent up to the *same bijection* used for the initial configurations. We prove that by constructing a square commutative diagram for heaps and stores, similar to Figure 2.10. (We explain the construction for

heaps, stores are treated analogously).[17] In the figure, heaps $\mu_1$ and $\mu_2$ are from the initial configurations of the lemma, while $\mu_1'$ and $\mu_2'$ are the heaps from the final configurations. Since the initial configurations are $L$-equivalent, we have the vertical edge on the left, i.e., $\mu_1 \approx_L^{\beta} \mu_2$, by assumption. Instead, the horizontal edges of the square are obtained by applying store and heap confinement to each individual program execution. In particular, the lemma shows that the initial and final heaps in these executions are $L$-equivalent up to appropriate identity bijections, i.e., $\mu_1 \approx_L^{\iota_{|\mu_1|}} \mu_1'$ and $\mu_2 \approx_L^{\iota_{|\mu_2|}} \mu_2'$, which ignore new heap allocations, as explained above. Therefore, the bijection that relates the final heaps from the vertical edge on the right of the square can be simplified. Specifically, the identity bijections *cancel out* in the composition with the bijection from the initial $L$-equivalence relation, i.e., $\iota_{|\mu_2|} \circ \beta \circ \iota_{|\mu_1|}^{-1} = \beta$, and so the final heaps remain related up to the same bijection $\beta$ used for the initial heaps, i.e., $\mu_1' \approx_L^{\beta} \mu_2'$. In the following, we use dot-notation to access individual elements of a configuration, e.g., we write $c.pc$ to extract the program counter $pc$ from the configuration $c = \langle \_, \_, pc, \_ \rangle$.

**Lemma 3.6** ($L$-Equivalence Preservation in Secret Contexts)**.** For all valid inputs $\vdash \mathbf{Valid}(c_1, \theta_1)$ and $\vdash \mathbf{Valid}(c_2, \theta_2)$ and bijections $\beta$, such that $c_1.pc \not\sqsubseteq L$, $c_2.pc \not\sqsubseteq L$, and $c_1 \approx_L^{\beta} c_2$, if $c_1 \Downarrow^{\theta_1} c_1'$ and $c_2 \Downarrow^{\theta_2} c_2'$, then $c_1' \approx_L^{\beta} c_2'$.

We now adapt the proof of $L$-equivalence in *public contexts*. The references allocated by the program considered in this lemma are observable by the attacker and so we have to construct an appropriate bijection to relate their addresses in the final configurations. However, since these allocations occur at run time and may depend on the program inputs, we cannot, in general, predict the exact bijection needed to relate them. Therefore, the final bijection is *existentially quantified* in the lemma, so that the right bijection can be precisely constructed, depending on the program execution, in each case of the proof itself. Unfortunately, existential quantification also complicates the proof, which now involves

[17] Heaps and stores are very often treated homogeneously in proofs, therefore our mechanized proof scripts pairs them up in a *program state* helper data structure to shorten some proofs.

reasoning about terms that are $L$-equivalent, but up to arbitrary bijections, and so may not be combined together. To solve this issue, the lemma additionally requires to prove an ordering invariant about these bijections, i.e., that the final bijection *extends* the initial bijection. Intuitively, this extra property solves the issue described above because $L$-equivalent relations up to "smaller" bijections can be lifted to "larger" bijections via *weakening* (Property 5.4), and so combined together.

It is worth pointing out that this issue is not specific to $\lambda^{dCG}$: the security analysis of $\lambda^{dFG}$ presents the same issue, which we solve in the same way in Section 2.4. However, only the analysis of the monadic fragment of the semantics of $\lambda^{dCG}$ is affected by the issue described above: the final bijection does not need to be quantified when reasoning about the pure fragment of the semantics. This is because pure reductions cannot allocate new references (they do not even have access to a heap) and so the same bijection that relates their input values can also relate their output values. This simplifies our formal analysis, as previous lemmas and their proofs are largely unaffected by the addition of bijections. For example, to adapt Lemma 3.3, i.e., *Pure $L$-Equivalence Preservation*, for flow-sensitive references, we *only* need to add the bijection $\beta$ to the $L$-equivalent relations in the statement of lemma: the proof requires no changes.

**Lemma 3.7** (Pure $L$-Equivalence Preservation)**.** For all expressions $e_1 \equiv_\alpha e_2$, bijections $\beta$, and environments $\theta_1 \approx_L^\beta \theta_2$, if $e_1 \Downarrow^{\theta_1} v_1$ and $e_2 \Downarrow^{\theta_2} v_2$, then $v_1 \approx_L^\beta v_2$.

The separation between pure and monadic semantics fragments of $\lambda^{dCG}$ simplifies the $L$-equivalence preservation of monadic computations as well. This is because most of the rules of this fragment of the semantics only include pure reductions and so intermediate $L$-equivalent results can be combined directly, i.e., no weakening is required in most cases.

**Lemma 3.8** ($L$-Equivalence Preservation in Public Contexts)**.** For all valid initial inputs $\vdash \mathbf{Valid}(c_1, \theta_1)$ and $\vdash \mathbf{Valid}(c_2, \theta_2)$ and bijections $\beta$, such that $c_1.pc = c_2.pc \sqsubseteq L$, $c_1 \approx_L^\beta c_2$, and $\theta_1 \approx_L^\beta \theta_2$, if $c_1 \Downarrow^{\theta_1} c_1'$ and $c_2 \Downarrow^{\theta_2} c_2'$, then there exists an extended bijection $\beta' \supseteq \beta$ such that $c_1' \approx_L^{\beta'} c_2'$.

Finally, we combine $L$-equivalence preservation in public and secret contexts and prove *termination-insensitive non-interference* for $\lambda^{dCG}$ extended with flow-sensitive references.

**Theorem 4** ($\lambda^{dCG}$-TINI with Bijections)**.** For all valid inputs $\vdash \mathbf{Valid}(c_1, \theta_1)$ and $\vdash \mathbf{Valid}(c_2, \theta_2)$ and bijections $\beta$, such that $c_1 \approx_L^{\beta} c_2, \theta_1 \approx_L^{\beta} \theta_2$, if $c_1 \Downarrow^{\theta_1} c_1'$, and $c_2 \Downarrow^{\theta_2} c_2'$, then there exists an extended bijection $\beta' \supseteq \beta$, such that $c_1' \approx_L^{\beta'} c_2'$.

**Conclusion.** At this point, we have formalized two calculi—$\lambda^{dFG}$ and $\lambda^{dCG}$—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 $\lambda^{dFG}$ and optional in $\lambda^{dCG}$—with significant implications for the programming model and usability. Second, they differ in the nature of the program counter, *flow-insensitive* in $\lambda^{dFG}$ and *flow-sensitive* in $\lambda^{dCG}$. Third, they differ in the way they deal with side-effects—$\lambda^{dCG}$ allows side-effectful computations exclusively inside the monad, while $\lambda^{dFG}$ is *impure*, i.e., any $\lambda^{dFG}$ 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 (Schmitz *et al.*, 2018; Buiras *et al.*, 2015; Jaskelioff and Russo, 2011; Tsai *et al.*, 2007; Russo *et al.*, 2009; Russo, 2015) 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 section we discuss our verified artifacts of $\lambda^{dFG}$ and $\lambda^{dCG}$ and compare their mechanized proofs of non-interference. Then, in Section 5 and 6 we show that—despite their differences—these two calculi are, in fact, equally expressive.

# 4

## Verified Artifacts

We now discuss our verified artifacts, in which we model $\lambda^{dFG}$ and $\lambda^{dCG}$ and provide machine-checked proofs of their security guarantees. We have formalized these languages using Agda (Norell, 2009; Bove *et al.*, 2009), a dependently typed functional language and an interactive proof assistant based on intuitionistic type theory. In our proof scripts, we embed the syntax, the type system, and the semantics of $\lambda^{dFG}$ and $\lambda^{dCG}$ into Agda data types, where we leverage dependent types to maintain additional assumptions about terms and judgments. For example, we use *well-typed syntax* and typed DeBrujin indexes to ensure that expressions, values, and environments are intrinsically well-scoped and well-typed (Abel *et al.*, 2019), which in turn let us define *type-preserving* big-step semantics judgments. In the following, we analyze our artifact and find that the security proofs for $\lambda^{dFG}$ are longer (between 43% and 74%) than those for $\lambda^{dCG}$. These empirical results suggest that reasoning about the security of coarse-grained IFC languages is easier than for fine-grained languages.

**Table 4.1:** The table reports the size (Agda LOC excluding blank lines and comments) of the proof scripts that formalize $\lambda^{dFG}$ and $\lambda^{dCG}$ and their security proofs with flow-insensitive references (FI) and with also flow-sensitive references (FI + FS).

| | FI | | FI + FS | |
|---|---|---|---|---|
| | $\lambda^{dFG}$ | $\lambda^{dCG}$ | $\lambda^{dFG}$ | $\lambda^{dCG}$ |
| Types | 15 | 16 | 16 | 17 |
| Syntax | 82 | 85 | 86 | 93 |
| Semantics | 137 | 134 | 164 | 162 |
| Valid | - | - | 230 | 198 |
| $L$-Equivalence | 143 | 138 | 212 | 217 |
| Security | 222 | 155 | 409 | 235 |
| Lattice | 148 | | 148 | |
| Store | 152 | | 112 | |
| Memory | | | 153 | |
| Heap | - | | 337 | |
| Bijection | - | | 478 | |
| Other | 66 | | 485 | |
| Total | 965 | 894 | 2,830 | 2,635 |

## 4.1 Artifact Analysis

Table 4.1 summarizes the size of our proof scripts. In the table, we report the number of lines of Agda code needed to formalize different parts of the fine- and coarse-grained languages ($\lambda^{dFG}$ and $\lambda^{dCG}$) featuring only flow-insensitive references (FI) and also with the addition of flow-sensitive references (FI + FS).[1]

The bottom part of the table lists parts of the formalization that are shared and reused for both languages, where categories Store, Memory and Heap include also definitions and proofs relative to their

[1] FI is the companion artifact of the conference version of this work (Vassena *et al.*, 2019), available at https://hub.docker.com/r/marcovassena/granularity. We developed artifact FI + FS by adding flow-sensitive references to artifact FI. The extended artifact is available at https://hub.docker.com/r/marcovassena/granularity-ftpl.

**Table 4.2:** The table reports the size of the main parts of the formal security analysis (Security) of $\lambda^{dFG}$ and $\lambda^{dCG}$ with and without flow-sensitive references. Confinement refers to the store and heap confinement lemma, Secret Context and Public Context to $L$-equivalence preservation in secret and public contexts, respectively, and Pure Preservation to $L$-equivalence preservation for the pure semantics fragment of $\lambda^{dCG}$.

| | FI | | FI + FS | |
|---|---|---|---|---|
| | $\lambda^{dFG}$ | $\lambda^{dCG}$ | $\lambda^{dFG}$ | $\lambda^{dCG}$ |
| Confinement | 51 | 24 | 89 | 44 |
| Secret Context | 13 | 23 | 16 | 29 |
| Pure Preservation | - | 35 | - | 35 |
| Public Context | 138 | 46 | 273 | 85 |
| Tini | 9 | 17 | 10 | 21 |
| Other | 11 | 10 | 21 | 21 |
| Total | 222 | 155 | 409 | 235 |

$L$-equivalence relations and valid judgments.[2] The line counts for basic definitions (i.e., Types, Syntax, Semantics, and $L$-equivalence) are roughly the same for the two languages, in both artifacts. The only significant difference is in Security, which represents the size of the scripts that state and prove the main security lemmas and theorems of the languages. In particular, we find that the security proofs in $\lambda^{dFG}$ are about 43% longer than in $\lambda^{dCG}$ in the artifact with only flow-insensitive references (FI) and 74% longer for the languages extended with flow-sensitive references (FI + FS). To understand better the reason behind this gap, we compare Security in more detail in Table 4.2, which reports the size of the following lemmas and theorems:

- ▷ Store and Heap Confinement
- ▷ $L$-Equivalence Preservation in Secret Context
- ▷ Pure $L$-Equivalence Preservation
- ▷ $L$-Equivalence Preservation in Public Context
- ▷ Termination-Insensitive Non-Interference

[2]We report only one number for Store and Memory because they are defined together in the same module in FI, but in separate modules in FI + FS. In the second artifact, Memory and Heap are also defined by instantiating a generic container data structure for labeled data, which is counted in Other.

First of all, we observe that in $\lambda^{dFG}$ the Confinement lemma is about twice as long as in $\lambda^{dCG}$. This is because side-effects can occur in every reduction rule in $\lambda^{dFG}$, while most constructs in $\lambda^{dCG}$ are side-effect free and so we simply have to consider fewer cases in the second proof. For the languages extended with flow-sensitive reference (FI + FS), these proofs roughly *double* in size, as they additionally need to consider the validity of program configurations in order to reason about $L$-equivalence up to bijection. The size of Secret Context and Tini is modest in both variants of each languages. Compared to $\lambda^{dFG}$, $\lambda^{dCG}$ requires *twice* as many lines of code for these results simply because for this language each result has to be stated and proved *twice*, once for the forcing semantics and once for the thunk semantics.

Public Context is the lemma where we observe the greatest gap in the number of lines needed in $\lambda^{dFG}$ and $\lambda^{dCG}$. In artifact FI, we find that Public Context for $\lambda^{dFG}$ is 70% longer than Public Context and Pure Preservation for $\lambda^{dCG}$ combined, and 128% longer when considering also flow-sensitive references (FI + FS). Since all values are intrinsically labeled in $\lambda^{dFG}$, those proofs have to *explicitly* (i) reason about the sensitivity of program inputs and intermediate values in most cases of the proof, and (ii) rule out implicit flows for control-flow constructs. In contrast, most constructs in $\lambda^{dCG}$ are security unaware and so evaluated by the pure fragment of the semantics, where the security analysis is straightforward. In particular, since pure reductions cannot perform side-effects or inspect labeled data, no implicit flows can arise in the proof of Pure Preservation, which follows by simple induction (35 LOC). Only in the proof of Public Context for $\lambda^{dCG}$ we need to reason about data flows explicitly using security labels, but even there implicit flows are less problematic, thanks to the monadic structure of computations. The line counts for Public Context in FI + FS show that the extra complexity of dealing with valid assumptions increases the size of the proofs in both languages, which grows by 85% for $\lambda^{dCG}$ and 98% for $\lambda^{dFG}$.[3] Not only the proofs of $\lambda^{dFG}$ are longer than $\lambda^{dCG}$, but they are also more complicated because they

[3]Importantly though, the addition of flow-sensitive references did not require any change in the proof of Pure Preservation.

often must (i) combine data related up to different bijections, and (ii) propagate the assumption that program configurations are valid in inductive cases. This results in an increased number of calls to helper lemmas and properties in $\lambda^{dFG}$, compared to $\lambda^{dCG}$. For example, the proof for PUBLIC CONTEXT in $\lambda^{dFG}$ requires **22** calls to *weakening* (Property 5.4) and **52** calls to *valid invariant* (Property 4), while the same proof in $\lambda^{dCG}$ requires only **4** calls in total.

# 5

# Fine- to Coarse-Grained Program Translation

This section presents a provably semantics-preserving program translation from the fine-grained dynamic IFC calculus $\lambda^{dFG}$ to the coarse-grained calculus $\lambda^{dCG}$. At a high level, the translation performs two tasks (i) it embeds the *intrinsic* label annotation of $\lambda^{dFG}$ values into an *explicitly* labeled $\lambda^{dCG}$ value via the **Labeled** type constructor and (ii) it restructures $\lambda^{dFG}$ *side-effectful* expressions into *monadic operations* inside the **LIO** monad.

## 5.1 Types and Values

Our type-driven approach starts by formalizing this intuition in the function $\langle\!\langle \cdot \rangle\!\rangle$, which maps the $\lambda^{dFG}$ type $\tau$ to the corresponding $\lambda^{dCG}$ type $\langle\!\langle \tau \rangle\!\rangle$ (see Figure 5.1a). The function is defined by induction on types and recursively adds the **Labeled** type constructor to each existing $\lambda^{dFG}$ type constructor. For the function type $\tau_1 \to \tau_2$, the result is additionally monadic, i.e., $\langle\!\langle \tau_1 \rangle\!\rangle \to \mathbf{LIO}\ \langle\!\langle \tau_2 \rangle\!\rangle$. This is because the function's body in $\lambda^{dFG}$ may have side-effects. Furthermore, the translation for references types $\mathbf{Ref}\ s\ \tau$ preserves the sensitivity tag of the reference, i.e., $\mathbf{Ref}\ s\ \langle\!\langle \tau \rangle\!\rangle$.

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

**(a)** Types.

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

**(b)** Values.

**Figure 5.1:** Translation from $\lambda^{dFG}$ to $\lambda^{dCG}$.

The translation for values (Figure 5.1b) is straightforward. Each $\lambda^{dFG}$ label tag becomes the label annotation in a $\lambda^{dCG}$ labeled value. The translation is homomorphic in the constructors on raw values. The translation converts a $\lambda^{dFG}$ function closure into a $\lambda^{dCG}$ thunk 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., $\langle\!\langle \theta \rangle\!\rangle = \lambda x.\langle\!\langle \theta(x) \rangle\!\rangle$. Finally, the translation preserves the memory address and the label for flow-insensitive references, i.e., $\langle\!\langle n_{\ell} \rangle\!\rangle = n_{\ell}$, and the heap address for flow-sensitive references, i.e., $\langle\!\langle n \rangle\!\rangle = n$.

## 5.2 Expressions

We show the translation of $\lambda^{dFG}$ expressions to $\lambda^{dCG}$ monadic thunks in Figure 5.2. We use the standard **do** notation for readability.[1] 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

[1] Syntax **do** $x \leftarrow e_1; e_2$ desugars to **bind**$(e_1, x.e_2)$ and syntax $e_1; e_2$ to **bind**$(e_1, \_.e_2)$.

$$\langle\!\langle () \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{return}(()))$$

$$\langle\!\langle \ell \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{return}(\ell))$$

$$\langle\!\langle (\lambda x.e) \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{return}(\lambda x.\langle\!\langle e \rangle\!\rangle))$$

$$\begin{aligned}&\langle\!\langle \mathbf{inl}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\\ &\quad lv \leftarrow \langle\!\langle e \rangle\!\rangle\\ &\quad \mathbf{return}(\mathbf{inl}(lv)))\end{aligned}$$

$$\begin{aligned}&\langle\!\langle \mathbf{inr}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\\ &\quad lv \leftarrow \langle\!\langle e \rangle\!\rangle\\ &\quad \mathbf{return}(\mathbf{inr}(lv)))\end{aligned}$$

$$\begin{aligned}&\langle\!\langle (e_1, e_2) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\\ &\quad lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle\\ &\quad lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle\\ &\quad \mathbf{return}(lv_1, lv_2))\end{aligned}$$

$$\langle\!\langle x \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{unlabel}(x))$$

$$\begin{aligned}&\langle\!\langle e_1\ e_2 \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\\ &\quad lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle\\ &\quad lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle\\ &\quad v_1 \leftarrow \mathbf{unlabel}(lv_1)\\ &\quad lv \leftarrow v_1\ lv_2\\ &\quad \mathbf{unlabel}(lv))\end{aligned}$$

$$\begin{aligned}&\langle\!\langle \mathbf{case}(e, x.e_1, x.e_2) \rangle\!\rangle =\\ &\quad \mathbf{toLabeled}(\ \mathbf{do}\\ &\quad\quad lv \leftarrow \langle\!\langle e \rangle\!\rangle\\ &\quad\quad v \leftarrow \mathbf{unlabel}(lv)\\ &\quad\quad lv' \leftarrow \mathbf{case}(v, x.\langle\!\langle e_1 \rangle\!\rangle, x.\langle\!\langle e_2 \rangle\!\rangle)\\ &\quad\quad \mathbf{unlabel}(lv'))\end{aligned}$$

$$\begin{aligned}&\langle\!\langle \mathbf{fst}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\\ &\quad lv \leftarrow \langle\!\langle e \rangle\!\rangle\\ &\quad v \leftarrow \mathbf{unlabel}(lv)\\ &\quad \mathbf{unlabel}(\mathbf{fst}(v)))\end{aligned}$$

$$\begin{aligned}&\langle\!\langle \mathbf{snd}(e) \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\\ &\quad lv \leftarrow \langle\!\langle e \rangle\!\rangle\\ &\quad v \leftarrow \mathbf{unlabel}(lv)\\ &\quad \mathbf{unlabel}(\mathbf{snd}(v)))\end{aligned}$$

$$\begin{aligned}&\langle\!\langle e_1\ \sqsubseteq^?\ e_2 \rangle\!\rangle = \mathbf{toLabeled}(\ \mathbf{do}\\ &\quad lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle\\ &\quad lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle\\ &\quad lu \leftarrow \mathbf{toLabeled}(\mathbf{return}(()))\\ &\quad v_1 \leftarrow \mathbf{unlabel}(lv_1)\\ &\quad v_2 \leftarrow \mathbf{unlabel}(lv_2)\\ &\quad \mathbf{return}(\ \mathbf{if}\\ &\quad\quad v_1\ \sqsubseteq^?\ v_2\\ &\quad\quad\quad \mathbf{then\ inl}(lu)\\ &\quad\quad\quad \mathbf{else\ inr}(lu))\end{aligned}$$

$$\begin{aligned}&\langle\!\langle \mathbf{taint}(e_1, e_2) \rangle\!\rangle =\\ &\quad \mathbf{toLabeled}(\ \mathbf{do}\\ &\quad\quad lv_1 \leftarrow \langle\!\langle e_1 \rangle\!\rangle\\ &\quad\quad v_1 \leftarrow \mathbf{unlabel}(lv_1)\\ &\quad\quad \mathbf{taint}(v_1)\\ &\quad\quad lv_2 \leftarrow \langle\!\langle e_2 \rangle\!\rangle\\ &\quad\quad \mathbf{unlabel}(lv_2))\end{aligned}$$

$$\begin{aligned}&\langle\!\langle \mathbf{labelOf}(e) \rangle\!\rangle =\\ &\quad \mathbf{toLabeled}(\ \mathbf{do}\\ &\quad\quad lv \leftarrow \langle\!\langle e \rangle\!\rangle\\ &\quad\quad \mathbf{labelOf}(lv))\end{aligned}$$

$$\langle\!\langle \mathbf{getLabel} \rangle\!\rangle = \mathbf{toLabeled}(\mathbf{getLabel}))$$

**Figure 5.2:** Translation from $\lambda^{dFG}$ to $\lambda^{dCG}$ (expressions).

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 $\lambda^{dFG}$ 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 $\lambda^{dFG}$ 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 $\lambda^{dFG}$ rules [Inl,Inr,Pair].[2] For example, the $\lambda^{dFG}$ reduction $((), ()) \Downarrow_L^{\varnothing} ((()^L, ()^L)^L$ maps to a $\lambda^{dCG}$ term that reduces to **Labeled** $L$ (**Labeled** $L$ (), **Labeled** $L$ ()) when started with program counter label $L$.

The translation of variables gives some insight into how the $\lambda^{dCG}$ floating-label mechanism can simulate $\lambda^{dFG}$'s tainting approach. First, the type-driven approach set out in Figure 5.1a 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 $\theta$ and translated separately, e.g., if $\theta(x) = r^\ell$ in $\lambda^{dFG}$, then $x$ gets bound to $\langle\!\langle r^\ell \rangle\!\rangle =$ **Labeled** $\ell$ $\langle\!\langle r \rangle\!\rangle$ when translated to $\lambda^{dCG}$. 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 $\lambda^{dFG}$ rule [Var], i.e., $x \Downarrow_{pc}^{\theta[x \mapsto r^\ell]} r^{pc \sqcup \ell}$.

[2] We name a variable $lv$ if it gets bound to a labeled value, i.e., to indicate that the variable has type **Labeled** $\tau$.

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., $v_1 \leftarrow \mathbf{unlabel}(lv_1)$.[3] 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 $\mathbf{toLabeled}(\cdot)$ wraps it again right away). The translation of case analysis is analogous. The translation of pair projections first converts the $\lambda^{dFG}$ pair into a computation that gives a $\lambda^{dCG}$ labeled pair of labeled values, say $\mathbf{Labeled}\ \ell\ (\mathbf{Labeled}\ \ell_1\ \langle\!\langle r_1 \rangle\!\rangle, \mathbf{Labeled}\ \ell_2\ \langle\!\langle r_2 \rangle\!\rangle)$ 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 $\lambda^{dFG}$ 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 $\lambda^{dCG}$ 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 $\lambda^{dFG}$ uses the corresponding $\lambda^{dCG}$ primitive applied on the corresponding labeled value, say $\mathbf{Labeled}\ \ell\ \langle\!\langle r \rangle\!\rangle$, obtained from the translated expression. Notice that $\mathbf{labelOf}(\cdot)$ taints the program counter label in $\lambda^{dCG}$, which rises to $pc \sqcup \ell$, so the code just described results in $\mathbf{Labeled}\ (pc \sqcup \ell)\ \ell$, which corresponds to the translation of the result in $\lambda^{dFG}$, i.e., $\langle\!\langle \ell^{\ell} \rangle\!\rangle = \mathbf{Labeled}\ \ell\ \ell$ because $pc \sqcup \ell \equiv \ell$, since $pc \sqsubseteq \ell$ from Property 1. The translation of **getLabel** follows naturally

[3] 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* $\lambda^{dFG}$ rule [App] for function application.

$$\frac{\Gamma \setminus \overline{x} \vdash e : \tau}{\Gamma \vdash \mathbf{wken}(\overline{x}, e) : \tau}\ (\textsc{WkenType}) \qquad \frac{e \Downarrow^{\theta \setminus \overline{x}} v}{\mathbf{wken}(\overline{x}, e) \Downarrow^{\theta} v}\ (\textsc{Wken})$$

**Figure 5.3:** Typing and semantics rules of **wken** for $\lambda^{dCG}$.

by simply wrapping $\lambda^{dCG}$'s **getLabel** inside a **toLabeled**$(\cdot)$, which correctly returns the program counter label labeled with itself, i.e., **Labeled** *pc pc*.

### 5.2.1 Note on Environments and Weakening

The semantics rules of $\lambda^{dFG}$ and $\lambda^{dCG}$ feature an environment $\theta$ 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 $\lambda^{dFG}$ 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 $e$, see the formal typing rule [WkenType] in Figure 5.3. At run-time, the expression **wken**$(\overline{x}, e)$ evaluates $e$ in an environment from which variables $\overline{x}$ have been dropped, so that they do not get captured in any closure created during the execution of $e$. Rule [Wken] is part of the pure semantics of $\lambda^{dCG}$—the semantics of $\lambda^{dFG}$ includes an analogous rule.

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

**Figure 5.4:** Translation $\lambda^{dFG}$ to $\lambda^{dCG}$ (references).

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).

## 5.3 References

Figure 5.4 shows the program translation of $\lambda^{dFG}$ primitives that access the store and the heap via references. Notice that these translations are the same for flow-insensitive and flow-sensitive references: they replicate the behavior of $\lambda^{dFG}$ primitives for each kind of reference using the corresponding primitives of $\lambda^{dCG}$. Below we explain the translation for flow-insensitive references, the discussion for flow-sensitive references is analogous.

The translation of $\lambda^{dFG}$ values wraps references in $\lambda^{dCG}$ labeled values (Figure 5.1b), so the translations of Figure 5.4 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 ($\mathbf{new}(lv)$). The $\lambda^{dCG}$ rule [New] (Figure

3.4) assigns the label of the translated content to the new reference, which also gets labeled with the original program counter label[4], just as in the $\lambda^{dFG}$ rule [New] (Figure 2.4). In $\lambda^{dFG}$, rule [Read] reads from a reference ${n_\ell}^{\ell'}$ at security level $\ell'$ that points to the $\ell$-labeled memory, and returns the content $\Sigma(\ell)[n]^{\ell \sqcup \ell'}$ at level $\ell \sqcup \ell'$. Similarly, the translation creates a **toLabeled**$(\cdot)$ block that executes to get a labeled reference $lr =$ **Labeled** $\ell'$ $n_\ell$, extracts the reference $n_\ell$ ($r \leftarrow$ **unlabel**$(lr)$) tainting the program counter label with $\ell'$, and then reads the reference's content further tainting the program counter label with $\ell$ as well. The code that translates and updates a reference consists of two **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 ($r \leftarrow$ **unlabel**$(lr)$) and updates it ($r := lv$). The second block, **toLabeled**(**return**()), returns unit at security level $pc$, i.e., **Labeled** $pc$ (), similar to the $\lambda^{dFG}$ rule [Write]. The translation of **labelOfRef**$(e)$ extracts the reference and projects its label via the $\lambda^{dCG}$ primitive **labelOfRef**$(\cdot)$, which additionally taints the program counter with the label itself, similar to the $\lambda^{dFG}$ rule [LabelOfRef].

## 5.4 Correctness

In this section, we establish some desirable properties of the $\lambda^{dFG}$-to-$\lambda^{dCG}$ 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 (Rajani and Garg, 2018; Barthe *et al.*, 2007).

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

**Lemma 5.1** (Type Preservation)**.** Given a well-typed $\lambda^{dFG}$ expression, i.e., $\Gamma \vdash e : \tau$, the translated $\lambda^{dCG}$ expression is also well-typed, i.e., $\langle\!\langle \Gamma \rangle\!\rangle \vdash \langle\!\langle e \rangle\!\rangle : \mathbf{LIO}\ \langle\!\langle \tau \rangle\!\rangle$.

[4]The nested block does not execute any **unlabel**$(\cdot)$ nor **taint**$(\cdot)$.

*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 $\lambda^{dFG}$ programs when translated and executed with $\lambda^{dCG}$ semantics (under a translated environment). In the theorem below, the translation of stores, memories, and heaps 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 $\ell$-labeled memory $M$, and $\langle\!\langle [\,] \rangle\!\rangle = [\,]$ and $\langle\!\langle r^{\ell} : \mu \rangle\!\rangle = \langle\!\langle r^{\ell} \rangle\!\rangle : \langle\!\langle \mu \rangle\!\rangle$ for heaps $\mu$. 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 $\lambda^{dFG}$ (by means of primitive **toLabeled**$(\cdot)$).

**Theorem 5** (Semantics Preservation of $\langle\!\langle \cdot \rangle\!\rangle : \lambda^{dFG} \to \lambda^{dCG}$ **).**
For all well-typed $\lambda^{dFG}$ programs $e$, if $\langle \Sigma, \mu, e \rangle \Downarrow^{\theta}_{pc} \langle \Sigma', \mu', v \rangle$, then $\langle \langle\!\langle \Sigma \rangle\!\rangle, \langle\!\langle \mu \rangle\!\rangle, pc, \langle\!\langle e \rangle\!\rangle \rangle \Downarrow^{\langle\!\langle \theta \rangle\!\rangle} \langle \langle\!\langle \Sigma' \rangle\!\rangle, \langle\!\langle \mu' \rangle\!\rangle, pc, \langle\!\langle v \rangle\!\rangle \rangle$.

*Proof.* By induction on the given evaluation derivation using basic properties of the security lattice and of the translation function.[5] □

### 5.5 Recovery of Non-Interference

We conclude this section by constructing a proof of termination-insensitive non-interference for $\lambda^{dFG}$ (Theorem 2) from the corresponding theorem for $\lambda^{dCG}$ (Theorem 4), using the semantics preserving translation (Theorem 5), together with a property that the translation preserves $L$-equivalence (Lemma 5.2), the validity of references (Lemma 5.4), as well as a property to recover source $L$-equivalence from target $L$-equivalence (Lemma 5.3). This result ensures that the meaning of labels is preserved by the translation (Rajani and Garg, 2018; Barthe *et al.*, 2007). 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

[5] In our mechanized proofs, this proof also requires the (often used) axiom of functional extensionality.

translation would not be meaningful from the perspective of security.[6] The following lemma shows that the translation of $\lambda^{dFG}$ *initial* configurations, defined as $\langle\!\langle c \rangle\!\rangle^{pc} = \langle \langle\!\langle \Sigma \rangle\!\rangle, \langle\!\langle \mu \rangle\!\rangle, pc, \langle\!\langle e \rangle\!\rangle \rangle$ if $c = \langle \Sigma, \mu, e \rangle$, preserves $L$-equivalence by lifting $L$-equivalence from source to target and back. Since the translation preserves the address of references and the size of the heap, the lemma relates the target configurations using the same bijection that relates the source configurations.

**Lemma 5.2** ($\langle\!\langle \cdot \rangle\!\rangle$ preserves $\approx_L^\beta$)**.** For all program counter labels $pc$ and bijections $\beta$, $c_1 \approx_L^\beta c_2$ if and only if $\langle\!\langle c_1 \rangle\!\rangle^{pc} \approx_L^\beta \langle\!\langle c_2 \rangle\!\rangle^{pc}$.

*Proof.* By definition of $L$-equivalence for initial configurations in both directions (Sections 2.2 and 3.2), using injectivity of the translation function, i.e., if $\langle\!\langle e_1 \rangle\!\rangle \equiv_\alpha \langle\!\langle e_2 \rangle\!\rangle$ then $e_1 \equiv_\alpha e_2$, in the *if* direction, and by mutually proving similar lemmas for all categories. □

The following lemma recovers $L$-equivalence of *source* final configurations by back-translating $L$-equivalence of *target* final configurations. We define the translation for $\lambda^{dFG}$ *final* configurations as $\langle\!\langle c \rangle\!\rangle^{pc} = \langle \langle\!\langle \Sigma \rangle\!\rangle, \langle\!\langle \mu \rangle\!\rangle, pc, \langle\!\langle v \rangle\!\rangle \rangle$ if $c = \langle \Sigma, \mu, v \rangle$.

**Lemma 5.3** ($\approx_L^\beta$ recovery via $\langle\!\langle \cdot \rangle\!\rangle$)**.** Let $c_1 = \langle \Sigma_1, \mu_1, {r_1}^{\ell_1} \rangle$, $c_2 = \langle \Sigma_2, \mu_2, {r_2}^{\ell_2} \rangle$ be $\lambda^{dFG}$ final configurations. For all bijections $\beta$ and program counter label $pc$, such that $pc \sqsubseteq \ell_1$ and $pc \sqsubseteq \ell_2$, if $\langle\!\langle c_1 \rangle\!\rangle^{pc} \approx_L^\beta \langle\!\langle c_2 \rangle\!\rangle^{pc}$ then $c_1 \approx_L^\beta c_2$.

*Proof.* By case analysis on the $L$-equivalence relation of the target final configurations, two cases follow. First, through $\langle\!\langle c_1 \rangle\!\rangle \approx_L^\beta \langle\!\langle c_2 \rangle\!\rangle$ we recover $L$-equivalence of the source stores, i.e., $\Sigma_1 \approx_L^\beta \Sigma_2$, from that of the target stores, i.e., $\langle\!\langle \Sigma_1 \rangle\!\rangle \approx_L^\beta \langle\!\langle \Sigma_2 \rangle\!\rangle$, and of the source heaps, i.e., $\mu_1 \approx_L^\beta \mu_2$, from the target heaps $\langle\!\langle \mu_1 \rangle\!\rangle \approx_L^\beta \langle\!\langle \mu_2 \rangle\!\rangle$. Then, the program counter in the target configurations is either (i) *above* the attacker's level $[\textsc{Pc}_H]$, i.e., $pc \not\sqsubseteq L$, and the source values are $L$-equivalent, i.e., ${r_1}^{\ell_1} \approx_L^\beta {r_2}^{\ell_2}$ by

[6] 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 (Rajani and Garg, 2018; Barthe *et al.*, 2007).

rule [VALUE$_H$] applied to $\ell_1 \not\sqsubseteq L$ and $\ell_2 \not\sqsubseteq L$ (from $pc \not\sqsubseteq L$ and, respectively, $pc \sqsubseteq \ell_1$ and $pc \sqsubseteq \ell_2$), or (ii) *below* the attacker's level [PC$_L$], i.e., $pc \sqsubseteq L$, then $\langle\!\langle {r_1}^{\ell_1} \rangle\!\rangle \approx_L^\beta \langle\!\langle {r_2}^{\ell_2} \rangle\!\rangle$ and the source values are $L$-equivalent, i.e., ${r_1}^{\ell_1} \approx_L^\beta {r_2}^{\ell_2}$, by Lemma 5.2 for values. □

Recall that non-interference for $\lambda^{dCG}$ extended with flow-sensitive (Theorem 4) requires side conditions about the validity of program inputs, which must contain only valid heap addresses (see the $\vdash \mathbf{Valid}(c, \theta)$ judgment in Figure 3.7). Thus, to recover non-interference for $\lambda^{dFG}$ through non-interference for $\lambda^{dCG}$, we need to instantiate these judgments in the proof. Luckily, non-interference for $\lambda^{dFG}$ also requires similar side conditions (Figure 2.9), so we can fulfill these assumptions by lifting the valid judgment for the source inputs into a valid judgment for the translated inputs.

**Lemma 5.4** ($\langle\!\langle \cdot \rangle\!\rangle$ preserves $\vdash \mathbf{Valid}(\cdot)$)**.** For all program counters $pc$, initial configurations $c$ and environments $\theta$, if $\vdash \mathbf{Valid}(c, \theta)$, then $\vdash \mathbf{Valid}(\langle\!\langle c \rangle\!\rangle^{pc}, \langle\!\langle \theta \rangle\!\rangle)$.

Finally, we recover termination insensitive non-interference for $\lambda^{dFG}$ through our semantics preserving translation to $\lambda^{dCG}$. Notice that the theorem statement below is identical to Theorem 2, what is new is the proof of the theorem, which relies on non-interference for $\lambda^{dCG}$ and our verified translation.

**Theorem 6** ($\lambda^{dFG}$-TINI with Bijections via $\langle\!\langle \cdot \rangle\!\rangle$)**.** For all program counter labels $pc$ and valid inputs $\vdash \mathbf{Valid}(c_1, \theta_1)$ and $\vdash \mathbf{Valid}(c_2, \theta_2)$, such that $c_1 \approx_L^\beta c_2$, $\theta_1 \approx_L^\beta \theta_2$, if $c_1 \Downarrow_{pc}^{\theta_1} c_1'$, and $c_2 \Downarrow_{pc}^{\theta_2} c_2'$, then there exists an extended bijection $\beta' \supseteq \beta$, such that $c_1' \approx_L^{\beta'} c_2'$.

*Proof.* We start by applying the fine to coarse grained program translation to the initial configurations and environments. By Theorem 5 (semantics preservation), we derive the corresponding $\lambda^{dCG}$ reductions, i.e., $\langle\!\langle c_1 \rangle\!\rangle^{pc} \Downarrow^{\langle\!\langle \theta_1 \rangle\!\rangle} \langle\!\langle c_1' \rangle\!\rangle^{pc}$ and $\langle\!\langle c_2 \rangle\!\rangle^{pc} \Downarrow^{\langle\!\langle \theta_2 \rangle\!\rangle} \langle\!\langle c_2' \rangle\!\rangle^{pc}$. Then, we lift $L$-equivalence of the initial configurations and environments from *source* to *target*, i.e., from $c_1 \approx_L^\beta c_2$ to $\langle\!\langle c_1 \rangle\!\rangle^{pc} \approx_L^\beta \langle\!\langle c_2 \rangle\!\rangle^{pc}$ and from $\theta_1 \approx_L^\beta \theta_2$ to $\langle\!\langle \theta_1 \rangle\!\rangle \approx_L^\beta \langle\!\langle \theta_2 \rangle\!\rangle$ (Lemma 5.2), and similarly we lift the validity judgments (Lemma 5.4), i.e., $\vdash \mathbf{Valid}(\langle\!\langle c_1 \rangle\!\rangle, \langle\!\langle \theta_1 \rangle\!\rangle)$ and $\vdash \mathbf{Valid}(\langle\!\langle c_2 \rangle\!\rangle, \langle\!\langle \theta_2 \rangle\!\rangle)$

from $\vdash \mathbf{Valid}(c_1, \theta_1)$ and $\vdash \mathbf{Valid}(c_2, \theta_2)$, respectively. Then, we apply $\lambda^{dCG}$-*TINI with Bijections* (Theorem 4) and obtain $L$-equivalence of the *target* final configurations, i.e., $\langle\!\langle c_1' \rangle\!\rangle^{pc} \approx_L^{\beta'} \langle\!\langle c_2' \rangle\!\rangle^{pc}$ for some bijection $\beta' \supseteq \beta$. Finally, we recover $L$-equivalence of the final configurations from *target* to *source*, i.e., from $\langle\!\langle c_1' \rangle\!\rangle^{pc} \approx_L^{\beta'} \langle\!\langle c_2' \rangle\!\rangle^{pc}$ to $c_1' \approx_L^{\beta'} c_2'$, via Lemma 5.3, applied to $c_1' = \langle \_, \_, {r_1}^{\ell_1} \rangle$ and $c_2' = \langle \_, \_, {r_2}^{\ell_2} \rangle$, and where $pc \sqsubseteq \ell_1$ and $pc \sqsubseteq \ell_2$ by Property 1 applied to the source reductions, i.e., $c_1 \Downarrow_{pc}^{\theta_1} c_1'$ and $c_2 \Downarrow_{pc}^{\theta_2} c_2'$. □

# 6

# Coarse- to Fine-Grained Program Translation

We now show a verified program translation in the opposite direction—from the coarse grained calculus $\lambda^{dCG}$ to the fine grained calculus $\lambda^{dFG}$. The translation in this direction is more involved—a program in $\lambda^{dFG}$ contains strictly more information than its counterpart in $\lambda^{dCG}$, 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 $\lambda^{dFG}$ calculus is designed to compute and attach labels at run-time. On the other hand, the translation cannot conservatively under-approximate the values of labels[1]—$\lambda^{dCG}$ and $\lambda^{dFG}$ 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 $\lambda^{dCG}$ and (ii) lets the $\lambda^{dFG}$ semantics compute the remaining label

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

annotations automatically—we account for those labels with a *cross-language* relation that represents semantic equivalence between $\lambda^{dFG}$ and $\lambda^{dCG}$ modulo extra annotations (Section 6.4). The fact that the source program in $\lambda^{dCG}$ cannot inspect those labels—they have no value counterpart in the source $\lambda^{dCG}$ 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 $\lambda^{dCG}$ program into the label annotation of the $\lambda^{dFG}$ value that results from executing the translated program. For example, if a $\lambda^{dCG}$ monadic expression starts with program counter label $pc$ and results in some value, say **true**, and final program counter $pc'$, then the translated $\lambda^{dFG}$ 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 $\lambda^{dFG}$ at run-time in a controlled way.

## 6.1 Types and Values

**Types.** The $\lambda^{dCG}$-to-$\lambda^{dFG}$ translation follows the same type-driven approach used in the other direction, starting from the function $[\![\cdot]\!]$ in Figure 6.1a, that translates a $\lambda^{dFG}$ type $\tau$ into the corresponding $\lambda^{dCG}$ type $[\![\tau]\!]$. The translation is defined by induction on $\tau$ and preserves all the type constructors standard types. Only the cases corresponding to $\lambda^{dCG}$-specific types are interesting. In particular, it converts *explicitly* labeled types, i.e., **Labeled** $\tau$, to a standard pair type in $\lambda^{dFG}$, i.e., $\mathscr{L} \times [\![\tau]\!]$, where the first component is the label and the second component the content of type $\tau$. Type **LIO** $\tau$ becomes a *suspension* in $\lambda^{dFG}$, i.e., the function type $\mathbf{unit} \to [\![\tau]\!]$ 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 6.1b. Notice that the translation is indexed by the program

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

**(a)** Types.

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

**(b)** Values.

**Figure 6.1:** Translation from $\lambda^{dCG}$ to $\lambda^{dFG}$ (part I).

counter label (the translation is written $[\![v]\!]^{pc}$), which converts the $\lambda^{dCG}$ value $v$ in scope of a computation protected by security level $pc$ to the corresponding fully label-annotated $\lambda^{dFG}$ value. The translation is pretty straightforward and uses the program counter label to tag each value, following the $\lambda^{dCG}$ principle that the program counter label protects every value in scope that is not explicitly labeled. The translation converts a $\lambda^{dCG}$ function closure into a corresponding $\lambda^{dFG}$ function closure by translating the body of the function to a $\lambda^{dFG}$ 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 $\lambda^{dFG}$ 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 $\mathbf{Labeled}\ \ell\ v$, into a labeled pair at security level $pc$, i.e., $(\ell^{\ell}, [\![v]\!]^{\ell})^{pc}$. The pair consists of the label $\ell$ tagged with itself, and the value translated at a security level equal to the label

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

**(a)** Expressions.

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

**(b)** Thunks.

**Figure 6.2:** Translation from $\lambda^{dCG}$ to $\lambda^{dFG}$ (part II).

annotation, i.e., $[\![v]\!]^{\ell}$. Notice that tagging the label with itself allows us to translate the $\lambda^{dCG}$ (label introspection) primitive $\mathbf{labelOf}(\cdot)$ by simply projecting the first component, thus preserving the label and its security level across the translation.

## 6.2 Expressions and Thunks

The translation of pure expressions (Figure 6.2a) 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 $\lambda$-abstraction, i.e., $\lambda\_.[\![t]\!]$, (the name of the variable is again irrelevant, thus we omit it as explained above), and translates the thunk itself according to the definition in Figure 6.2b. 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,[2] 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] (Figure 2.2) 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 $\lambda^{dCG}$'s [Unlabel] rule in Figure 3.3b. 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 $\lambda^{dCG}$ 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 $\lambda^{dFG}$. Primitive $\mathbf{labelOf}(e)$ simply projects the first component of the pair that encodes the labeled value in $\lambda^{dFG}$ as explained above. Lastly, **getLabel** in $\lambda^{dCG}$ maps directly to **getLabel** in $\lambda^{dFG}$—rule [GetLabel] in $\lambda^{dCG}$ simply returns the program counter label and does not raise its value, so it corresponds exactly to rule [GetLabel] in $\lambda^{dFG}$, which returns label $pc$ at security level $pc$. Similarly, $\mathbf{taint}(e)$ translates to $\mathbf{taint}([\![ e ]\!], ())$, since rule [Taint] in $\lambda^{dCG}$ taints the program counter with the label that $e$ evaluates to, say $\ell$ and returns unit with program counter label equal to $pc \sqcup \ell$, which corresponds to the result of the translated program, i.e., $()^{pc \sqcup \ell}$.

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

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

**Figure 6.3:** Translation from $\lambda^{dCG}$ to $\lambda^{dFG}$ (references).

## 6.3 References

Figure 6.3 shows the translation of primitives that access the store and the heap via references. Like the translation in the opposite direction, these translations work homogeneously for flow-insensitive and flow-sensitive references, therefore we focus our discussion on flow-insensitive references. Since $\lambda^{dCG}$'s rule [New] in Figure 3.4 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 $\lambda^{dFG}$'s rule [New] in Figure 2.4 assigns the correct label to the new reference. Due to its *fine-grained* precision, $\lambda^{dFG}$ might have labeled the content with a different label that is less sensitive than the explicit label that *coarsely* approximates the security level in $\lambda^{dCG}$. Similarly, when updating a reference we translate the new labeled value into a pair, i.e., $\mathbf{let}\ x = [\![e_2]\!]$, and taint the content with the label of the labeled value projected from the pair, i.e., $\mathbf{taint}(\mathbf{fst}(x), \mathbf{snd}(x))$.[3] The translation of the primitives that read and query the label of a reference is trivial.

[3]Technically, tainting is redundant for flow-insensitive references, i.e., the translation $[\![e_1]\!] := \mathbf{snd}([\![e_2]\!])$ from the conference version of this work would preserve the semantics for references $\Gamma \vdash e_1 : \mathbf{Ref}\ I\ \tau$. Since flow-insensitive references have a *fixed* label, rule [Write] in both $\lambda^{dFG}$ and $\lambda^{dCG}$ accepts values less sensitive than the reference and stores them in the memory labeled like the reference. In contrast, the label of flow-sensitive references is not fixed and tainting is necessary to set the correct label for the value written in the heap.

### 6.4 Cross-Language Equivalence Relation

When a $\lambda^{dCG}$ program is translated to $\lambda^{dFG}$ via the program translation described above, the $\lambda^{dFG}$ result contains strictly more information than the original $\lambda^{dCG}$ result. This happens because the semantics of $\lambda^{dFG}$ tracks flows of information at fine granularity, in contrast with $\lambda^{dCG}$, which instead coarsely approximates the security level of all values in scope of a computation with the program counter label. When translating a $\lambda^{dCG}$ program, we can capture this condition precisely for input values $\theta$ by *homogeneously* tagging all standard (unlabeled) values with the initial program counter label, i.e., $[\![\theta]\!]^{pc}$. However, a $\lambda^{dCG}$ 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 $\lambda^{dFG}$ 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 $\lambda^{dFG}$ result will *not* be homogeneously labeled with the final program counter label of the $\lambda^{dCG}$ computation, i.e., if a $\lambda^{dCG}$ program terminates with value $v$ and program counter label $pc'$, the translated $\lambda^{dFG}$ program does *not* necessarily result in $[\![v]\!]^{pc'}$. The following example illustrates this issue, which we then address via a *cross-language* equivalence relation that correctly approximates the additional labels computed by $\lambda^{dFG}$.

**Example 6.1.** Consider the execution of $\lambda^{dCG}$ program $e = \mathbf{taint}(H); \mathbf{return}(x)$, i.e., $\langle \Sigma, L, \mathbf{taint}(H); \mathbf{return}(x) \rangle \Downarrow^{[x \mapsto \mathbf{true}]} \langle \Sigma, H, \mathbf{true} \rangle$, which taints the program counter label with $H$, and then returns $\mathbf{true} = \mathbf{inl}(())$ and the store $\Sigma$ unchanged.

Let $[\![e]\!]$ be the expression obtained by applying the program translation from Figure 6.2 to the example program:

$$
\begin{array}{l}
[\![e]\!] = \lambda\_. \\
\quad \mathbf{let}\ y = \mathbf{taint}(H, ())\ \mathbf{in} \\
\quad\quad \mathbf{taint}(\mathbf{labelOf}(y), x)
\end{array}
$$

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

$$n \downarrow\!\approx_{pc} n \text{ (Ref-FS)} \qquad \frac{v_1 \downarrow\!\approx_{pc} v_1'}{\mathbf{inl}(v_1) \downarrow\!\approx_{pc} \mathbf{inl}(v_1')} \text{(Inl)} \qquad \frac{v_2 \downarrow\!\approx_{pc} v_2'}{\mathbf{inr}(v_2) \downarrow\!\approx_{pc} \mathbf{inr}(v_2')} \text{(Inr)}$$

$$\frac{v_1 \downarrow\!\approx_{pc} v_1' \qquad v_2 \downarrow\!\approx_{pc} v_2'}{(v_1, v_2) \downarrow\!\approx_{pc} (v_1', v_2')} \text{(Pair)} \qquad \frac{\theta_1 \downarrow\!\approx_{pc} \theta_2}{(x.[\![e]\!], \theta_1) \downarrow\!\approx_{pc} (x.e, \theta_2)} \text{(Fun)}$$

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

**Figure 6.4:** Cross-language value equivalence modulo label annotations.

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, i.e., $x \mapsto [\![\mathbf{true}]\!]^L = \mathbf{inl}(()^L)^L = \mathbf{true}^L$, we do *not* obtain the translated result homogeneously labeled with $H$:

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

In particular, $\lambda^{dFG}$ preserves the public label tag on data nested inside the left injection, i.e., $()^L$ in $\mathbf{inl}(()^L)^H$ above. This happens because $\lambda^{dFG}$'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 $\lambda^{dFG}$ value.[4] Technically, we use this equivalence in the semantics preservation theorem in Section 6.5, which *existentially* quantifies over the result of the translated $\lambda^{dFG}$ program, but guarantees that it is semantically equivalent to the result obtained in the source program.

Concretely, for a $\lambda^{dFG}$ value $v_1$ and a $\lambda^{dCG}$ 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 6.4 formalizes this intuition by means of two mutually inductive relations, one for $\lambda^{dFG}$ values and one for $\lambda^{dFG}$ raw values. In particular, rule [Value] relates $\lambda^{dFG}$ value ${r_1}^{\ell_1}$ and $\lambda^{dCG}$ 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,Ref-FS]), requires the same injection for values of the sum type ([Inl,Inr]) and requires related components 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 $\lambda^{dFG}$ function body $x.[\![e]\!]$ (resp. thunk body $\_.[\![t]\!]$) is obtained from the $\lambda^{dCG}$ function body $e$ (resp. thunk $t$) via the program translation defined above. Lastly, rule [Labeled] relates a $\lambda^{dCG}$ labeled value **Labeled** $\ell$ $v_1$ to a pair $(\ell^{\ell}, v_2)$, consisting of the label $\ell$ protected by itself in the first component and value $v_2$ related with the content $v_1$ at security level $\ell$ ($v_1 \downarrow\approx_{\ell} v_2$) in the second component. This rule follows the principle of **LIO** that for explicitly labeled values, the label annotation represents an upper

[4] 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.

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 $\ell$-labeled memories relate their contents respectively at security level $\ell$, i.e., $[\,] \downarrow\!\approx [\,]$ and $(r : M_1) \downarrow\!\approx (v : M_2)$ iff $r \downarrow\!\approx_\ell v$ and $M_1 \downarrow\!\approx M_2$ for $\lambda^{dFG}$ and $\lambda^{dCG}$ memories $M_1, M_2 : \mathit{Memory}\ \ell$. Heaps are also related pointwise, i.e., $[\,] \downarrow\!\approx [\,]$ and $(r^\ell : \mu_1) \downarrow\!\approx (\textbf{Labeled}\ \ell\ v : \mu_2)$ iff $r \downarrow\!\approx_\ell v$ and $\mu_1 \downarrow\!\approx \mu_2$, where values at corresponding positions must additionally have the same label. Lastly, we lift the relation to initial and final configurations.

**Definition 4** (Cross-Language Equivalence of Configurations)**.** For all initial and final configurations:

- $\langle \Sigma_1, \mu_1, [\![ e ]\!]\ () \rangle \downarrow\!\approx \langle \Sigma_2, \mu_2, pc, e \rangle$ iff $\Sigma_1 \downarrow\!\approx \Sigma_2$ and $\mu_1 \downarrow\!\approx \mu_2$,
- $\langle \Sigma_1, \mu_1, [\![ t ]\!] \rangle \downarrow\!\approx \langle \Sigma_2, \mu_2, pc, t \rangle$ iff $\Sigma_1 \downarrow\!\approx \Sigma_2$ and $\mu_1 \downarrow\!\approx \mu_2$.
- $\langle \Sigma_1, \mu_1, r^{pc} \rangle \downarrow\!\approx \langle \Sigma_2, \mu_2, pc, v \rangle$ iff $\Sigma_1 \downarrow\!\approx \Sigma_2$, $\mu_1 \downarrow\!\approx \mu_2$, and $r \downarrow\!\approx_{pc} v$.

First, the relation requires the stores and heaps of initial and final configurations to be related. Additionally, for initial configurations, the relation requires the $\lambda^{dFG}$ code to be obtained from the $\lambda^{dCG}$ expression (resp. thunk) via the program translation function $[\![\cdot]\!]$ defined above (similar to rules [Fun] and [Thunk] in Figure 6.4). Furthermore, in the first case (expressions), the relation additionally forces the translated suspension $[\![ e ]\!]$ by applying it to $()$, so that when the $\lambda^{dFG}$ security monitor executes the translated program, it obtains the result that corresponds to the $\lambda^{dCG}$ monadic program $e$. Finally, in the definition for final configurations, the security level of the final $\lambda^{dFG}$ result must match the program counter label $pc$ of the final $\lambda^{dCG}$ configuration, and the final $\lambda^{dCG}$ result must correspond to the $\lambda^{dFG}$ 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 cross-language equivalence relation that will be useful later. The following property allows instantiating the semantics preservation theorem with the $\lambda^{dCG}$ initial configuration. The translation for initial configurations is per-component, i.e., $[\![\langle \Sigma, \mu, pc, t \rangle]\!] = \langle [\![\Sigma]\!], [\![\mu]\!], [\![ t ]\!] \rangle$ and forcing for suspensions, i.e., $[\![\langle \Sigma, \mu, pc, e \rangle]\!] = \langle [\![\Sigma]\!], [\![\mu]\!], [\![ e ]\!]\ () \rangle$, pointwise for stores, i.e., $[\![\Sigma]\!] = \lambda \ell . [\![\Sigma(\ell)]\!]$, memories, i.e., $[\![ [\,] ]\!] = [\,]$

and $[\![v : M]\!] = [\![v]\!]^{\ell} : [\![M]\!]$ for $\ell$-labeled memory $M$, and heaps, i.e., $[\![[\,]]\!] = [\![[\,]]\!]$ and $[\![\mathbf{Labeled}\ \ell\ v : \mu]\!] = [\![v]\!]^{\ell} : [\![\mu]\!]$.

**Property 8** (Reflexivity of $\downarrow\approx$)**.** For all $\lambda^{dCG}$ 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 heaps $[\![\mu]\!] \downarrow\approx \mu$, for environments $[\![\theta]\!]^{pc} \downarrow\approx_{pc} \theta$, for values $[\![v]\!]^{pc} \downarrow\approx_{pc} v$, for any label $pc$. □

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

**Property 9** (Weakening)**.** For all labels $pc$ and $pc'$ such that $pc \sqsubseteq pc'$, and for all $\lambda^{dFG}$ raw values $r_1$, values $v_1$ and environments $\theta_1$, and $\lambda^{dCG}$ values $v_2$ and environments $\theta_2$:

- ▷ If $r_1 \downarrow\approx_{pc} v_2$ then $r_1 \downarrow\approx_{pc'} v_2$
- ▷ 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 cross-language equivalence relation. □

### 6.5 Correctness

With the help of the cross-language relation defined above, we can now state and prove that the $\lambda^{dCG}$-to-$\lambda^{dFG}$ 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 $\lambda^{dCG}$ program when executed under $\lambda^{dFG}$ semantics, with the same program counter label and translated input values. Since the translated $\lambda^{dFG}$ program computes strictly more information than the original $\lambda^{dCG}$ program, the theorem existentially quantify over the $\lambda^{dFG}$ result, but insists that it is semantically equivalent to the original $\lambda^{dCG}$ 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 6.1** (Type Preservation)**.** If $\Gamma \vdash e : \tau$ then $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$.

*Proof.* By induction on the typing judgment. □

Next, we prove semantics preservation of $\lambda^{dCG}$ 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 $\lambda^{dFG}$ and do not change the state in $\lambda^{dFG}$.

**Lemma 6.2** ($[\![\cdot]\!] : \lambda^{dCG} \to \lambda^{dFG}$ preserves Pure Semantics)**.** If $e \Downarrow^{\theta} v$ then for any program counter label $pc$, $\lambda^{dFG}$ store $\Sigma$, heap $\mu$, and environment $\theta'$ such that $\theta' \downarrow\!\approx_{pc} \theta$, there exists a raw value $r$, such that $\langle \Sigma, \mu, [\![e]\!] \rangle \Downarrow^{\theta'}_{pc} \langle \Sigma, \mu, r^{pc} \rangle$ 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 $\theta'$ in relation with the source environment $\theta$ 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 7** (Thunk and Forcing Semantics Preservation via $\downarrow\!\approx$)**.** For all $\lambda^{dFG}$ environments $\theta_1$, initial configurations $c_1$ and $\lambda^{dCG}$ environments $\theta_2$ and initial configuration $c_2$, such that $\theta_1 \downarrow\!\approx_{c_2.pc} \theta_2$ and $c_1 \downarrow\!\approx c_2$, if $c_2 \Downarrow^{\theta_2} c_2'$, then there exists a final configuration $c_1'$, such that $c_1 \Downarrow^{\theta_1}_{c_2.pc} c_1'$ and $c_1' \downarrow\!\approx c_2'$.

*Proof.* By mutual induction on the given derivation for expressions and thunks, using Lemma 6.2 for pure reductions, *Weakening* (Property 9), and basic properties for operations on related stores and heaps. □

We finally instantiate the semantics-preservation theorem (Theorem 7) with the translated input environment at security level $pc$, via *reflexivity* of the cross-language relation (Property 8).

**Corollary 1** (Semantics Preservation of $[\![\cdot]\!] : \lambda^{dCG} \to \lambda^{dFG}$)**.** For all $\lambda^{dCG}$ well-typed initial configurations $c_2$ and environments $\theta$, if $c_2 \Downarrow^{\theta} c_2'$, then there exists a final $\lambda^{dFG}$ configuration $c_1'$ such that $c_1' \downarrow\approx c_2'$ and $[\![ c_2 ]\!] \Downarrow_{pc}^{[\![\theta]\!]^{pc}} c_1'$, where $pc = c_2.pc$.

In the corollary above, the flow-sensitive program counter of the source $\lambda^{dCG}$ program gets encoded in the security level of the result of the $\lambda^{dFG}$ translated program. For example, if $\langle \Sigma_2, \mu_2, pc, e \rangle \Downarrow^{\theta} \langle \Sigma_2', \mu_2', pc', v \rangle$ then, by Corollary 1 and unrolling Definition 4, there exists a raw value $r$ at security level $pc'$, a store $\Sigma_1'$, and a heap $\mu_1'$ such that $\langle [\![\Sigma_2]\!], [\![\mu_2]\!], [\![e]\!]\ () \rangle \Downarrow_{pc}^{[\![\theta]\!]^{pc}} \langle \Sigma_1', \mu_1', r^{pc'} \rangle$, $r \downarrow\approx_{pc'} v$, $\Sigma_1' \downarrow\approx \Sigma_2'$, and $\mu_1' \downarrow\approx \mu_2'$.

### 6.6 Recovery of Non-Interference

Similarly to our presentation of Theorem 6 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 $\lambda^{dCG}$ through the program translation defined above, semantics preservation (Corollary 1), and $\lambda^{dFG}$ non-interference (Theorem 2). By reproving non-interference of the source language from the target language, we show that our program translation is authentic.

To prove this result, we first need to prove that the translation preserves $L$-equivalence (Lemma 6.3) and the validity of references (Lemma 6.7), as well as a property for recovering source $L$-equivalence from target $L$-equivalence through the cross-language relation (Lemma 6.6). The following lemma ensures that the translation of initial configurations preserves $L$-equivalence.

**Lemma 6.3** ($[\![\cdot]\!]$ preserves $\approx_L^{\beta}$)**.** For all bijections $\beta$ and initial configurations $c_1$ and $c_2$, if $c_1 \approx_L^{\beta} c_2$, then $[\![ c_1 ]\!] \approx_L^{\beta} [\![ c_2 ]\!]$.

*Proof.* By induction on the $L$-equivalence judgment and proving similar lemmas for all syntactic categories. □

The following lemmas recovers $\lambda^{dCG}$ $L$-equivalence from $\lambda^{dFG}$ $L$-equivalence using the cross-language equivalence relation for values and environments in public contexts.

**Lemma 6.4** ($\approx_L^\beta$ recovery from $\downarrow\approx_L$ for values and envs)**.** For all bijections $\beta$ and public program counter labels $pc \sqsubseteq L$, for all $\lambda^{dFG}$ values $v_1$, $v_2$, raw values $r_1$, $r_2$, environments $\theta_1$, $\theta_2$, and corresponding $\lambda^{dCG}$ values $v_1'$, $v_2'$, and environments $\theta_1'$, $\theta_2'$:

- If $v_1 \approx_L^\beta v_2$, $v_1 \downarrow\approx_{pc} v_1'$ and $v_2 \downarrow\approx_{pc} v_2'$, then $v_1' \approx_L^\beta v_2'$,
- If $r_1 \approx_L^\beta r_2$, $r_1 \downarrow\approx_{pc} v_1'$ and $r_2 \downarrow\approx_{pc} v_2'$, then $v_1' \approx_L^\beta v_2'$,
- If $\theta_1 \approx_L^\beta \theta_2$, $\theta_1 \downarrow\approx_{pc} \theta_1'$ and $\theta_2 \downarrow\approx_{pc} \theta_2'$, then $\theta_1' \approx_L^\beta \theta_2'$.

*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.[5] □

Next, we extend this result to program state, i.e., stores, memories, and heaps.

**Lemma 6.5** ($\approx_L^\beta$ recovery from $\downarrow\approx$ for state)**.** For all bijections $\beta$, $\lambda^{dFG}$ memories $M_1$, $M_2$, stores $\Sigma_1$, $\Sigma_2$, heaps $\mu_1$, $\mu_2$, and corresponding $\lambda^{dCG}$ memories $M_1'$, $M_2'$, stores $\Sigma_1'$, $\Sigma_2'$, heaps $\mu_1'$ and $\mu_2'$:

- If $M_1 \approx_L^\beta M_2$, $M_1 \downarrow\approx M_1'$ and $M_2 \downarrow\approx M_2'$, then $M_1' \approx_L^\beta M_2'$,
- If $\Sigma_1 \approx_L^\beta \Sigma_2$, $\Sigma_1 \downarrow\approx \Sigma_1'$ and $\Sigma_2 \downarrow\approx \Sigma_2'$, then $\Sigma_1' \approx_L^\beta \Sigma_2'$,
- If $\mu_1 \approx_L^\beta \mu_2$, $\mu_1 \downarrow\approx \mu_1'$ and $\mu_2 \downarrow\approx \mu_2'$, then $\mu_1' \approx_L^\beta \mu_2'$.

*Proof.* By induction on the $L$-equivalence relation and the cross-language equivalence relations and using Lemma 6.4. □

The next lemma lifts the previous lemmas to final configurations.

[5] Technically, the function $[\![\cdot]\!]$ presented in Section 6 is not injective. For example, consider the type translation function from Figure 6.1a: $[\![\mathbf{Labeled\ unit}]\!] = \mathscr{L} \times \mathbf{unit} = [\![\mathscr{L} \times \mathbf{unit}]\!]$ but $\mathbf{Labeled\ unit} \neq \mathscr{L} \times \mathbf{unit}$, and $[\![\mathbf{LIO\ unit}]\!] = \mathbf{unit} \to \mathbf{unit} = [\![\mathbf{unit} \to \mathbf{unit}]\!]$ but $\mathbf{LIO\ unit} \neq \mathbf{unit} \to \mathbf{unit}$. We make the translation injective by (i) adding a wrapper type $\mathbf{Id}\ \tau$ to $\lambda^{dFG}$, together with constructor $\mathbf{Id}(e)$, a deconstructor $\mathbf{unId}(e)$ and raw value $\mathbf{Id}(v)$, and (ii) tagging security-relevant types and terms with the wrapper, i.e., $[\![\mathbf{Labeled}\ \tau]\!] = \mathbf{Id}\ (\mathscr{L} \times [\![\tau]\!])$ and $\mathbf{LIO}\ \tau = \mathbf{Id\ unit} \to [\![\tau]\!]$. Adapting the translations in both directions is tedious but straightforward; we refer the interested reader to our mechanized proofs for details.

**Lemma 6.6** ($\approx_L^\beta$ recovery from $\downarrow\approx$)**.** Let $c_1$ and $c_2$ be $\lambda^{dFG}$ final configurations, let $c_1'$ and $c_2'$ be $\lambda^{dCG}$ final configurations. If $c_1 \approx_L^\beta c_2$, $c_1 \downarrow\approx c_1'$ and $c_2 \downarrow\approx c_2'$, then $c_1' \approx_L^\beta c_2'$.

*Proof.* Let $c_1 = \langle \Sigma_1, \mu_1, v_1 \rangle$, $c_2 = \langle \Sigma_2, \mu_2, v_2 \rangle$, $c_1' = \langle \Sigma_1', \mu_1', pc_1, v_1' \rangle$, $c_2' = \langle \Sigma_2', \mu_2', pc_2, v_2' \rangle$. From $L$-equivalence of the $\lambda^{dFG}$ final configurations, we have $L$-equivalence for the stores and the values, i.e., $\Sigma_1 \approx_L^\beta \Sigma_2$ and $v_1 \approx_L^\beta v_2$ from $c_1 \approx_L^\beta c_2$ (Section 2.2). Similarly, we derive cross-language equivalence relations for the components of the final configurations, i.e., respectively $\Sigma_1 \downarrow\approx \Sigma_1'$, $\mu_1 \downarrow\approx \mu_1'$, and $v_1 \downarrow\approx_{pc_1} v_1'$ from $c_1 \downarrow\approx c_2$, and $\Sigma_2 \downarrow\approx \Sigma_2'$, $\mu_2 \downarrow\approx \mu_2'$, and $v_2 \downarrow\approx_{pc_2} v_2'$ from $c_2 \downarrow\approx c_2'$ (Definition 4). First, the $\lambda^{dCG}$ stores and heaps are $L$-equivalent, i.e., $\Sigma_1' \approx_L^\beta \Sigma_2'$ and $\mu_1' \approx_L^\beta \mu_2'$ by Lemma 6.5 for stores and heaps, respectively. Then, two cases follow by case split on $v_1 \approx_L^\beta v_2$. Either (i) both label annotations on the values are not observable ([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 $c_1' \approx_L^\beta c_2'$ by rule [Pc$_H$] or (ii) the label annotations are equal and observable by the attacker ([Value$_L$]), i.e., $pc_1 \equiv pc_2 \sqsubseteq L$, then $v_1' \approx_L^\beta v_2'$ by Lemma 6.4 for values and $c_1' \approx_L^\beta c_2'$ by rule [Pc$_L$]. □

Before recovering non-interference, we show that the translation preserves validity of references, i.e., the assumption of the *TINI* theorem for $\lambda^{dFG}$ extended with flow-sensitive references.

**Lemma 6.7** ($[\![\cdot]\!]$ preserves $\vdash$ **Valid**$(\cdot)$)**.** For all initial configurations $c$ and environments $\theta$, if $\vdash$ **Valid**$(c, \theta)$, then $\vdash$ **Valid**$([\![c]\!], [\![\theta]\!]^{c.pc})$.

Finally, we combine these lemmas and prove *TINI* for $\lambda^{dCG}$ through our verified translation.

**Theorem 8** ($\lambda^{dCG}$-TINI with Bijections via $[\![\cdot]\!]$)**.** For all valid inputs $\vdash$ **Valid**$(c_1, \theta_1)$ and $\vdash$ **Valid**$(c_2, \theta_2)$ and bijections $\beta$, such that $c_1 \approx_L^\beta c_2$, $\theta_1 \approx_L^\beta \theta_2$, if $c_1 \Downarrow^{\theta_1} c_1'$, and $c_2 \Downarrow^{\theta_2} c_2'$, then there exists an extended bijection $\beta' \supseteq \beta$, such that $c_1' \approx_L^{\beta'} c_2'$.

*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 $\theta_1$ and $\theta_2$. Let

$pc$ be the initial program counter label common to configurations $c_1$ and $c_2$ (it is the same because $c_1 \approx_L c_2$). Then, *Semantics Preservation of* $[\![\cdot]\!]$ (Corollary 1) ensures that there exist two $\lambda^{dFG}$ 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 6.3, i.e., $[\![\theta_1]\!]^{pc} \approx_L^{\beta} [\![\theta_2]\!]^{pc}$ from $\theta_1 \approx_L^{\beta} \theta_2$ and $[\![ c_1 ]\!] \approx_L^{\beta} [\![ c_2 ]\!]$ from $c_1 \approx_L^{\beta} c_2$. Similarly, we lift the valid judgments for $\lambda^{dCG}$ inputs to $\lambda^{dFG}$ via Lemma 6.7, i.e., $\vdash \mathbf{Valid}([\![ c_1 ]\!], [\![\theta_1]\!])$ and $\vdash \mathbf{Valid}([\![ c_2 ]\!], [\![\theta_2]\!])$ from $\vdash \mathbf{Valid}(c_1, \theta_1)$ and $\vdash \mathbf{Valid}(c_2, \theta_2)$, respectively. Then, we apply $\lambda^{dFG}$*-TINI with Bijections* (Theorem 2) 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., $c_1'' \approx_L^{\beta'} c_2''$, up to some bijection $\beta' \supseteq \beta$. Then, we apply Lemma 6.6 to $c_1'' \approx_L^{\beta'} c_2''$, $c_1'' \downarrow\!\approx c_1'$, and $c_2'' \downarrow\!\approx c_2'$, and recover $L$-equivalence for the source configurations, i.e., $c_1' \approx_L^{\beta'} c_2'$, up to the same bijection $\beta' \supseteq \beta$. □

# 7

# Related work

## 7.1 Relative Expressiveness of IFC Systems

**Fine- and Coarse-Grained IFC.** 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 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. Furthermore, our

languages and translations support flow-sensitive references, which are not considered by Rajani *et al.* (2017), as the type-system of FG and CG is only flow-insensitive.

Our coarse-grained system $\lambda^{dCG}$ 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 $\lambda^{dFG}$ to $\lambda^{dCG}$. The reason for this is that our fine-grained system $\lambda^{dFG}$ is not a direct dynamic analogue of Rajani and Garg (2018)'s FG. In $\lambda^{dFG}$, 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 $\bot$. Hence, leaving aside the static-vs-dynamic difference, FG's labels are more precise than $\lambda^{dFG}$'s, and this makes Rajani and Garg (2018)'s FG to CG translation more difficult than our $\lambda^{dFG}$ to $\lambda^{dCG}$ translation. In fact, in earlier work, Rajani *et al.* (2017) introduced a different type system called $\text{FG}^{-}$, a static analogue of $\lambda^{dFG}$ 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).

**Flow-Insensitive and Flow-Sensitive IFC.** Researchers have explored the relative permissiveness of static and dynamic IFC systems with respect to the flow-sensitivity of the analysis. Hunt and Sands (2006) show that flow-sensitive *static* analysis are a natural generalization of flow-insensitive analysis. In the dynamic settings, Buiras *et al.* (2014) provide a semantics-preserving translation that embeds flow-sensitive references into flow-insensitive references in LIO (Stefan *et al.*, 2012). Their embedding tracks the mutable label of references through an extra level of indirection, i.e., they translate each flow-sensitive references into a flow-insensitive reference, which points to another flow-insensitive reference that stores the content. Interestingly, this result seems to

suggest that flow-sensitive references do not fundamentally increase the expressiveness and permissiveness of dynamic coarse-grained IFC systems. In contrast to our translations, however, their embedding is not local (or macro-expressible using the terminology of Felleisen (1991)): it relies on the flow-sensitive heap to assign a fixed label to flow-insensitive references.

Some works also study the relative permissiveness of static and dynamic IFC systems. As one would expected, dynamic flow-insensitive analysis are more permissive than their static counterpart (Sabelfeld and Russo, 2009). Russo and Sabelfeld (2010) show that, perhaps surprisingly, this is not the case for *flow-sensitive* analysis, i.e., purely dynamic and static flow-sensitive analysis are incomparable in terms of permissiveness, and propose a more permissive class of hybrid IFC monitors.

**Other IFC Techniques.** Balliu *et al.* (2017) develop a formal framework to study the soundness and permissiveness trade-offs of dynamic information-flow trackers, but their analysis is only with respect to explicit and implicit flows. Bielova and Rezk (2016a) compare dynamic and hybrid information-flow monitors for imperative languages with respect to soundness and transparency. *Secure multi-execution* (SME) is a dynamic IFC mechanisms that enforces security precisely, at the cost of executing a program multiple times (Devriese and Piessens, 2010). *Multiple facets* (MF) simulates SME through a single execution that maintains multiple views on data, but provides weaker security guarantees than SME (Austin and Flanagan, 2012). Bielova and Rezk (2016b) show that MF is not equivalent to SME (even when SME is relaxed to enforce the same security condition as MF) and adapt MF to provide the same guarantees of SME. Schmitz *et al.* (2018) unify MF and SME in a single framework that combines their best features by allowing programs to switch between the two mechanisms at run-time.

## 7.2 Coarse-Grained Dynamic IFC

Coarse-grained dynamic IFC systems are prevalent in security research in operating systems (Efstathopoulos *et al.*, 2005; Krohn *et al.*, 2007;

Zeldovich *et al.*, 2006). These ideas have also been successfully applied to other domains, e.g., the web (Giffin *et al.*, 2012; Stefan *et al.*, 2014; Yip *et al.*, 2009; Bauer *et al.*, 2015), mobile applications (Jia *et al.*, 2013; Nadkarni *et al.*, 2016), IoT (Fernandes *et al.*, 2016), and distributed systems (Zeldovich *et al.*, 2008; Pedersen and Chong, 2019; Cheng *et al.*, 2012). Our $\lambda^{dCG}$ calculus is based on LIO, a domain-specific language embedded in Haskell that rephrases OS-like IFC enforcement into a language-based setting (Stefan *et al.*, 2011; Stefan *et al.*, 2012). Heule *et al.* (2015) introduce a general framework for retrofitting coarse-grained IFC in any programming language in which external effects can be controlled. Co-Inflow (Xiang and Chong, 2021) extends Java with coarse-grained dynamic IFC, which is implemented via compilation, similar to our $\lambda^{dFG}$-to-$\lambda^{dCG}$ translation. 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.

## 7.3 Fine-Grained Dynamic IFC

The dangerous combination of highly dynamic scripting languages and third-party code in web pages (Nikiforakis *et al.*, 2012) and IoT platforms (Surbatovich *et al.*, 2017; Ahmadpanah *et al.*, 2021) has stirred a line of work on dynamic fine-grained IFC systems for JavaScript interpreters (Hedin *et al.*, 2014), engines (Bichhawat *et al.*, 2014b; Rajani *et al.*, 2015), and IoT apps (Bastys *et al.*, 2018). Our $\lambda^{dFG}$ calculus is inspired by the calculus of Austin and Flanagan (2009), which we have extended with flow-insensitive references and label introspection. In a follow up work, Austin and Flanagan (2010) replace the *no-sensitive upgrade* check with a *permissive upgrade* strategy, which tracks partially leaked data to ensure it is not completely leaked. Breeze is conceptually similar to our $\lambda^{dFG}$, except for the **taint**$(\cdot)$ primitive (Hritcu *et al.*, 2013). Breeze (Hritcu *et al.*, 2013), JSFlow (Hedin and Sabelfeld, 2012), and other dynamic fine-grained IFC languages (Bichhawat *et al.*, 2021; Austin *et al.*, 2017) feature exception handling primitives, which allow programs to recover from IFC violations without leaking data. Since LIO (Stefan *et al.*, 2017) features similar primitives, we believe that our results extend also to IFC languages with exceptions.

### 7.4 Label Introspection and Flow-Sensitive References

In general, dynamic fine-grained IFC systems often do not support label introspection, with Breeze (Hritcu *et al.*, 2013) as notable exception. Stefan *et al.* (2017) show that careless label introspection can leak data and discuss alternative flow-sensitive and flow-insensitive APIs (see Footnote 7). Xiang and Chong (2021) argue that the flow-sensitive API does not provide a usable programming model (because inspecting a label can unpredictably taint the program counter label) and use *opaque labeled values* instead. To support label introspection securely, our calculi protect each label with the label itself. Kozyri *et al.* (2019) generalizes this mechanism to chains of labels of arbitrary length (where each label defines the sensitivity of its predecessor) and study the trade-offs between permissiveness and storage.

Several dynamic fine-grained IFC systems support references with flow-sensitive labels (Hedin *et al.*, 2014; Austin and Flanagan, 2010; Austin and Flanagan, 2009; Bichhawat *et al.*, 2014b). This design choice, however, allows label changes to be exploited as a covert channel for information leaks (Russo and Sabelfeld, 2010; Austin and Flanagan, 2010; Austin and Flanagan, 2009). There are many approaches to preventing such leaks—from using static analysis techniques (Sabelfeld and Myers, 2006), to disallowing label upgrades depending on sensitive data (i.e., no-sensitive-upgrades (Zdancewic, 2002; Austin and Flanagan, 2009)), to avoiding branching on data whose labels have been upgraded (i.e., permissive-upgrades (Austin and Flanagan, 2010; Bichhawat *et al.*, 2021; Bichhawat *et al.*, 2014a)). Buiras *et al.* (2014) extend LIO with flow-sensitive references and explicitly protect the label of these references with the program counter label at creation time. The semantics of $\lambda^{dCG}$ avoids keeping track of this extra label by using the label of the value itself as a sound approximation (see rules [New-FS] and [Write-FS] in Figure 3.6b), which corresponds precisely to the semantics of $\lambda^{dFG}$.

### 7.5 Proof Techniques for Termination-Insensitive Non-Interference

Since Goguen and Meseguer (1982) introduced the notion of *non-interference*, different proof techniques for IFC languages have emerged.

Our proof technique based on *L-equivalence preservation* and *confinement* dates back to the seminal work by Volpano *et al.* (1996) and is similar to the proof by Austin and Flanagan (2009), although we do not make any assumptions about the heap allocator. Instead, our treatment of heap addresses is inspired by Banerjee and Naumann (2005): we extend the $L$-equivalence relation with a bijection, which accounts precisely for different, yet indistinguishable addresses. Although bijections can complicate the formal analysis, Vassena *et al.* (2017) argue that they can be avoided by partitioning data structures per security level (as we do here for the store). Moreover, the separation between pure computations and side-effects further simplifies the security analysis of monadic languages like $\lambda^{dCG}$ and, as we explain in Section 4, it leads to shorter proofs than in impure languages like $\lambda^{dFG}$. Hirsch and Cecchetti (2021) generalize this insight to other effects (non-termination and exceptions) through a new proof technique for pure languages that provide effects through a monad. In their fine-grained static IFC $\lambda$-calculus, Pottier and Simonet (2003) represent secret values and expressions explicitly, through a syntactic bracketed pair construct. This representation simplifies the non-interference proof (which is derived from subject reduction), but requires reasoning about a non-standard semantics.

**Logical Relations.** The first proofs of non interference that use logical relations are for the pure fragment of the fine-grained static IFC languages by Zdancewic (2002) and Heintze and Riecke (1998). Rajani *et al.* (2017) extend this proof technique for FG and CG, but their logical relation require step-indexed Kripke worlds (Birkedal *et al.*, 2011) to avoid circular arguments when reasoning about state. Recently, Gregersen *et al.* (2021) develop a mechanized semantic model based on logical relations on top of the Iris framework (Jung *et al.*, 2018) for an expressive fine-grained static IFC language. Proofs based on logical relations for stateful languages feature two types of logical relations: a *binary* relation for observable values (similar to $L$-equivalence), and a *unary* relation for secret values, which provides a semantics interpretation of the *confinement* lemma.

**PER and Parametricity.** Abadi *et al.* (1999) suggests a connection between parametricty and non-interference, which is formalized through a domain-theoretic semantics for the Dependency Core Calculus (DCC). Sabelfeld and Sands (2001) develop this idea further with a general semantics model of information flow based on partial equivalence relations (PER). Bowman and Ahmed (2015) prove non-interference for DCC from parametricity by proving that their translation from DCC into System F is fully abstract. Algehed and Bernardy (2019) simplifies this proof by embedding DCC into the Calculus of Construction and applies the same technique to derive a shorter proof for the core of LIO (Stefan *et al.*, 2017).

# 8

# Conclusion

This tutorial presents a detailed and homogeneous account of dynamic fine- and coarse-grained IFC security and unifies these paradigms, which were considered fundamentally at odds with respect to precision and permissiveness. To this end, we formalized two representative IFC languages that track information flows with fine and coarse granularity, established their security guarantees using standard proof techniques, and devised verified semantics- and security-preserving translations between them. These results formally establish a connection between dynamic fine- and coarse-grained enforcement for IFC, showing that these paradigms 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 a 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.

# References


Abadi, M., A. Banerjee, N. Heintze, and J. Riecke. (1999). "A Core Calculus of Dependency". In: *Proc. ACM Symp. on Principles of Programming Languages*. 147–160.

Abel, A., G. Allais, A. Hameer, B. Pientka, A. Momigliano, S. Schäfer, and K. Stark. (2019). "POPLMark reloaded: Mechanizing proofs by logical relations". *Journal of Functional Programming*. 29: e19. DOI: 10.1017/S0956796819000170.

Ahmadpanah, M. M., D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. (2021). "SandTrap: Securing JavaScript-driven Trigger-Action Platforms". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. 2899–2916. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadpanah.

Algehed, M. and J.-P. Bernardy. (2019). "Simple Noninterference from Parametricity". *Proc. ACM Program. Lang.* 3(ICFP). DOI: 10.1145/3341693.

Austin, T. H. and C. Flanagan. (2009). "Efficient Purely-Dynamic Information Flow Analysis". In: *Proc. of the 9th ACM Workshop on Programming Languages and Analysis for Security (PLAS '09)*.

Austin, T. H. and C. Flanagan. (2010). "Permissive Dynamic Information Flow Analysis". In: *Proc. of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. PLAS '10*.

Austin, T. H. and C. Flanagan. (2012). "Multiple Facets for Dynamic Information Flow". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '12*. Philadelphia, PA, USA: Association for Computing Machinery. 165–178. DOI: 10.1145/2103656.2103677.

Austin, T. H., T. Schmitz, and C. Flanagan. (2017). "Multiple Facets for Dynamic Information Flow with Exceptions". *ACM Trans. Program. Lang. Syst.* 39(3). DOI: 10.1145/3024086.

Balliu, M., D. Schoepe, and A. Sabelfeld. (2017). "We Are Family: Relating Information-Flow Trackers". In: *ESORICS*.

Banerjee, A. and D. A. Naumann. (2005). "Stack-based access control and secure information flow". *Journal Functional Programming*. 15(2): 131–177.

Barthe, G., T. Rezk, and A. Basu. (2007). "Security Types Preserving Compilation". *Computer Languages, Systems & Structures*. 33(2): 35–59. DOI: 10.1016/j.cl.2005.05.002.

Bastys, I., M. Balliu, and A. Sabelfeld. (2018). "If This Then What? Controlling Flows in IoT Apps". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. CCS '18*. Toronto, Canada: Association for Computing Machinery. 1102–1119. DOI: 10.1145/3243734.3243841.

Bauer, L., S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. (2015). "Run-time Monitoring and Formal Analysis of Information Flows in Chromium". In: *Proc. of the 22nd Annual Network & Distributed System Security Symposium*. Internet Society.

Bell, E. D. and J. L. La Padula. (1976). "Secure computer system: Unified exposition and Multics interpretation". Bedford, MA. URL: http://csrc.nist.gov/publications/history/bell76.pdf.

Bichhawat, A., V. Rajani, D. Garg, and C. Hammer. (2014a). "Generalizing Permissive-Upgrade in Dynamic Information Flow Analysis". In: *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security. PLAS'14*. Uppsala, Sweden: Association for Computing Machinery. 15–24. DOI: 10.1145/2637113.2637116.

Bichhawat, A., V. Rajani, D. Garg, and C. Hammer. (2014b). "Information Flow Control in WebKit's JavaScript Bytecode". In: *International Conference on Principles of Security and Trust (POST)*. 159–178.

Bichhawat, A., V. Rajani, D. Garg, and C. Hammer. (2021). "Permissive runtime information flow control in the presence of exceptions". *Journal of Computer Security*. 29: 361–401. DOI: 10.3233/JCS-211385.

Bielova, N. and T. Rezk. (2016a). "A Taxonomy of Information Flow Monitors". In: *Principles of Security and Trust*. Ed. by F. Piessens and L. Viganò. Berlin, Heidelberg: Springer Berlin Heidelberg. 46–67.

Bielova, N. and T. Rezk. (2016b). "Spot the Difference: Secure Multi-execution and Multiple Facets". In: *Computer Security – ESORICS 2016*. Cham: Springer International Publishing. 501–519.

Birkedal, L., B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. (2011). "Step-Indexed Kripke Models over Recursive Worlds". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '11*. Austin, Texas, USA: Association for Computing Machinery. 119–132. DOI: 10.1145/1926385.1926401.

Bove, A., P. Dybjer, and U. Norell. (2009). "A Brief Overview of Agda – A Functional Language with Dependent Types". In: *Theorem Proving in Higher Order Logics*. Ed. by S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel. Berlin, Heidelberg: Springer Berlin Heidelberg. 73–78.

Bowman, W. J. and A. Ahmed. (2015). "Noninterference for Free". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ICFP 2015*. Vancouver, BC, Canada: Association for Computing Machinery. 101–113. DOI: 10.1145/2784731.2784733.

Broberg, N., B. van Delft, and D. Sands. (2013). "Paragon for Practical Programming with Information-Flow Control". In: *Proc. of the 11th Asian Symposium on Programming Languages and Systems. APLAS '13*. 217–232.

Buiras, P., D. Stefan, and A. Russo. (2014). "On Dynamic Flow-Sensitive Floating-Label Systems". In: *Proc. of the 2014 IEEE 27th Computer Security Foundations Symposium. CSF '14*. Washington, DC, USA: IEEE Computer Society. 65–79. DOI: 10.1109/CSF.2014.13.

Buiras, P., D. Vytiniotis, and A. Russo. (2015). "HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ICFP 2015*. Vancouver, BC, Canada: Association for Computing Machinery. 289–301. DOI: 10.1145/2784731.2784758.

Cheng, W., D. R. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. (2012). "Abstractions for Usable Information Flow Control in Aeolus". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association. 139–151. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/cheng.

Devriese, D. and F. Piessens. (2010). "Noninterference through Secure Multi-execution". In: *Proc. of the 2010 IEEE Symposium on Security and Privacy. SP '10*. IEEE Computer Society.

Efstathopoulos, P., M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. (2005). "Labels and Event Processes in the Asbestos Operating System". In: *Proc. of the 20th ACM symp. on Operating systems principles. SOSP '05*.

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

Fernandes, E., J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. (2016). "FlowFence: Practical Data Protection for Emerging IoT Application Frameworks". In: *USENIX Security Symposium*. 531–548.

Giffin, D. B., A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. (2012). "Hails: Protecting Data Privacy in Untrusted Web Applications". In: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI '12*.

Goguen, J. and J. Meseguer. (1982). “Security Policies and Security Models”. In: *Proc. of IEEE Symposium on Security and Privacy*. IEEE Computer Society.

Gregersen, S. O., J. Bay, A. Timany, and L. Birkedal. (2021). “Mechanized Logical Relations for Termination-Insensitive Noninterference”. *Proc. ACM Program. Lang.* 5(POPL). DOI: 10.1145/3434291.

Hedin, D., 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)*.

Hedin, D. and D. Sands. (2006). “Noninterference in the presence of non-opaque pointers”. In: *Proc. of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press.

Hedin, D. and A. Sabelfeld. (2012). “Information-Flow Security for a Core of JavaScript”. In: *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society.

Heintze, N. and J. G. Riecke. (1998). “The SLam calculus: programming with secrecy and integrity”. In: *Proc. ACM Symp. on Principles of Programming Languages*. 365–377.

Heule, S., D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo. (2015). “IFC Inside: Retrofitting Languages with Dynamic Information Flow Control”. In: *Proc. of the Conference on Principles of Security and Trust (POST ’15)*. Springer.

Hirsch, A. K. and E. Cecchetti. (2021). “Giving Semantics to Program-Counter Labels via Secure Effects”. *Proc. ACM Program. Lang.* 5(POPL). DOI: 10.1145/3434316.

Hritcu, C., M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. (2013). “All Your IFCException Are Belong to Us”. In: *Proc. of the 2013 IEEE Symposium on Security and Privacy. SP ’13*. Washington, DC, USA: IEEE Computer Society. 3–17. DOI: 10.1109/SP.2013.10.

Hunt, S. and D. Sands. (2006). “On flow-sensitive security types”. In: *Conference record of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of programming languages. POPL ’06*. Charleston, South Carolina, USA: ACM. 79–90.

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

Jia, L., J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. (2013). "Run-Time Enforcement of Information-Flow Properties on Android". In: *Proc. of the 18th European Symposium on Research in Computer Security (ESORICS '13)*. Springer.

Jung, R., R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. (2018). "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". *Journal of Functional Programming*. 28: e20. DOI: 10.1017/S0956796818000151.

Kozyri, E., F. B. Schneider, A. Bedford, J. Desharnais, and N. Tawbi. (2019). "Beyond Labels: Permissiveness for Dynamic Information Flow Enforcement". In: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 351–35115. DOI: 10.1109/CSF.2019.00031.

Krohn, M., A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. (2007). *Information Flow Control for Standard OS Abstractions*. Stevenson, Washington, USA. DOI: 10.1145/1294261.1294293.

Myers, A. C., L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. (2006). *Jif 3.0: Java information flow*. URL: http://www.cs.cornell.edu/jif.

Nadkarni, A., B. Andow, W. Enck, and S. Jha. (2016). "Practical DIFC Enforcement on Android." In: *USENIX Security Symposium*. 1119–1136.

Nikiforakis, N., L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. (2012). "You Are What You Include: Large-Scale Evaluation of Remote Javascript Inclusions". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security. CCS '12*. Raleigh, North Carolina, USA: Association for Computing Machinery. 736–747. DOI: 10.1145/2382196.2382274.

Norell, U. (2009). "Dependently Typed Programming in Agda". In: *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Ed. by P. Koopman, R. Plasmeijer, and D. Swierstra. Berlin, Heidelberg: Springer Berlin Heidelberg. 230–266. DOI: 10.1007/978-3-642-04652-0_5.

Pedersen, M. V. and S. Chong. (2019). "Programming with Flow-Limited Authorization: Coarser is Better". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 63–78. DOI: 10.1109/EuroSP.2019.00015.

Pottier, F. and V. Simonet. (2003). "Information Flow Inference for ML". *ACM Trans. Program. Lang. Syst.* 25(1): 117–158. DOI: 10.1145/596980.596983.

Rajani, V., I. Bastys, W. Rafnsson, and D. Garg. (2017). "Type Systems for Information Flow Control: The Question of Granularity". *ACM SIGLOG News*. 4(1): 6–21. DOI: 10.1145/3051528.3051531.

Rajani, V., A. Bichhawat, D. Garg, and C. Hammer. (2015). "Information Flow Control for Event Handling and the DOM in Web Browsers". In: *2015 IEEE 28th Computer Security Foundations Symposium*. 366–379. DOI: 10.1109/CSF.2015.32.

Rajani, V. and D. 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.

Roy, I., D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. (2009). "Laminar: Practical Fine-grained Decentralized Information Flow Control". In: *Proc. of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '09*. Dublin, Ireland: ACM. 63–74. DOI: 10.1145/1542476.1542484.

Russo, A. (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.

Russo, A., K. Claessen, and J. Hughes. (2009). "A library for light-weight Information-Flow Security in Haskell". *ACM SIGPLAN Notices (HASKELL '08)*. 44(Jan.): 13. DOI: 10.1145/1543134.1411289.

Russo, A. and A. 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.

Sabelfeld, A. and A. Russo. (2009). "From dynamic to static and back: Riding the roller coaster of information-flow control research". In: *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (PSI '09). LNCS*. Springer-Verlag.

Sabelfeld, A. and A. C. Myers. (2006). "Language-based Information-flow Security". *IEEE J.Sel. A. Commun.* 21(1): 5–19. DOI: 10.1109/JSAC.2002.806121.

Sabelfeld, A. and D. Sands. (2001). "A Per Model of Secure Information Flow in Sequential Programs". *Higher Order Symbol. Comput.* 14(1): 59–91. DOI: 10.1023/A:1011553200337.

Schmitz, T., M. Algehed, C. Flanagan, and A. Russo. (2018). "Faceted Secure Multi Execution". In: *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security. CCS '18*. Toronto, Canada: ACM. 1617–1634. DOI: 10.1145/3243734.3243806.

Stefan, D., A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. (2012). "Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems". In: *International Conference on Functional Programming (ICFP)*. ACM SIGPLAN.

Stefan, D., A. Russo, D. Mazières, and J. C. Mitchell. (2017). "Flexible Dynamic Information Flow Control in the Presence of Exceptions". *Journal of Functional Programming*. 27.

Stefan, D., A. Russo, J. C. Mitchell, and D. Mazières. (2011). "Flexible Dynamic Information Flow Control in Haskell". In: *Proc. of the 4th ACM Symposium on Haskell. Haskell '11*. Tokyo, Japan: ACM. 95–106. DOI: 10.1145/2034675.2034688.

Stefan, D., E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. (2014). "Protecting Users by Confining JavaScript with COWL". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. OSDI'14*. Broomfield, CO: USENIX Association. 131–146. URL: http://dl.acm.org/citation.cfm?id=2685048.2685060.

Surbatovich, M., J. Aljuraidan, L. Bauer, A. Das, and L. Jia. (2017). "Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes". In: *Proceedings of the 26th International Conference on World Wide Web. WWW '17*. Perth, Australia: International World Wide Web Conferences Steering Committee. 1501–1510. DOI: 10.1145/3038912.3052709.

Tsai, T.-C., A. Russo, and J. Hughes. (2007). "A Library for Secure Multi-threaded Information Flow in Haskell". In: *Proc. of the 20th IEEE Computer Security Foundations Symposium (CSF'07)*. 187–202. DOI: 10.1109/CSF.2007.6.

Vassena, M. and A. Russo. (2016). "On Formalizing Information-Flow Control Libraries". In: *Proc. of the 2016 ACM Workshop on Programming Languages and Analysis for Security. PLAS '16*. Vienna, Austria: ACM. 15–28. DOI: 10.1145/2993600.2993608.

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

Vassena, M., A. Russo, D. Garg, V. Rajani, and D. Stefan. (2019). "From Fine- to Coarse-Grained Dynamic Information Flow Control and Back". *Proc. ACM Program. Lang.* 3(POPL). DOI: 10.1145/3290389.

Volpano, D., G. Smith, and C. Irvine. (1996). "A Sound Type System for Secure Flow Analysis". *J. Computer Security*. 4(3): 167–187.

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

Xiang, J. and S. Chong. (2021). "Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages". In: *Proceedings of the 2021 IEEE Symposium on Security and Privacy*. Piscataway, NJ, USA: IEEE Press.

Yang, J., K. Yessenov, and A. Solar-Lezama. (2012). "A Language for Automatically Enforcing Privacy Policies". In: *Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '12*. Philadelphia, PA, USA: ACM. 85–96. DOI: 10.1145/2103656.2103669.

Yip, A., N. Narula, M. Krohn, and R. Morris. (2009). "Privacy-preserving Browser-side Scripting with BFlow". In: *Proc. of the 4th ACM European Conference on Computer Systems. EuroSys '09.* ACM.

Zdancewic, S. A. (2002). "Programming Languages for Information Security". *PhD thesis.* Ithaca, NY, USA.

Zeldovich, N., S. Boyd-Wickizer, E. Kohler, and D. Mazières. (2006). "Making Information Flow Explicit in HiStar". In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7. OSDI '06.* Seattle, WA: USENIX Association. 19–19. URL: http://dl.acm.org/citation.cfm?id=1267308.1267327.

Zeldovich, N., S. Boyd-Wickizer, and D. Mazières. (2008). "Securing Distributed Systems with Information Flow Control". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation. NSDI'08.* San Francisco, California: USENIX Association. 293–308. URL: http://dl.acm.org/citation.cfm?id=1387589.1387610.