Simple Noninterference by Normalization

Nachiappan Valliappan[*]
Chalmers University of Technology
nacval@chalmers.se

Carlos Tomé Cortiñas[*]
Chalmers University of Technology
carlos.tome@chalmers.se

Abstract

Information-flow control (IFC) languages ensure programs preserve the confidentiality of sensitive data. *Noninterference*, the desired security property of such languages, states that public outputs of programs must not depend on sensitive inputs. In this paper, we show that noninterference can be proved using normalization. Unlike arbitrary terms, normal forms of programs are well-principled and obey useful syntactic properties—hence enabling a simpler proof of noninterference. Since our proof is syntax-directed, it offers an appealing alternative to traditional semantic based techniques to prove noninterference.

In particular, we prove noninterference for a static IFC calculus, based on Haskell's `seclib` library, using normalization. Our proof follows by straight-forward induction on the structure of normal forms. We implement normalization using *normalization by evaluation* and prove that the generated normal forms preserve semantics. Our results have been verified in the Agda proof assistant.

CCS Concepts • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*;

Keywords information-flow control, noninterference, normalization by evaluation

ACM Reference Format:
Nachiappan Valliappan and Carlos Tomé Cortiñas. 2019. Simple Noninterference by Normalization. In *Proceedings of ACM SIGPLAN Conference on Programming Languages (PLAS'19).* ACM, New York, NY, USA, 13 pages.

1 Introduction

Information-flow control (IFC) is a security mechanism which guarantees confidentiality of sensitive data by controlling how information is allowed to flow in a program. The guarantee that programs secured by an IFC system do not leak sensitive data is often proved using a property called *noninterference*. Noninterference ensures that an observer authorized to view the output of a program (pessimistically called the attacker) cannot infer any sensitive data handled by it. For example, suppose that the type $\mathsf{Int}_{\mathrm{H}}$ denotes a secret integer and $\mathsf{Bool}_{\mathrm{L}}$ denotes a public boolean. Now consider a program f with the following type:

$$f : \mathsf{Int}_{\mathrm{H}} \rightarrow \mathsf{Bool}_{\mathrm{L}}$$

For this program, noninterference ensures that f outputs the same boolean output for any given integer.

To prove noninterference, we must show that the public output of a program is not affected by varying the secret input. This has been achieved using many techniques including *term erasure* based on dynamic operational semantics [Li and Zdancewic 2010; Russo et al. 2009; Stefan et al. 2011; Vassena and Russo 2016], denotational semantics [Abadi et al. 1999; Kavvos 2019], and *parametricity* [Algehed and Bernardy 2019; Bowman and Ahmed 2015; Tse and Zdancewic 2004]. In this paper, we show that noninterference can also be proved by normalizing programs using the static or *residualising* semantics [Lindley 2005] of the language.

If a program returns the same output for any given input, it must be the case that it does not depend on the input to compute the output. Thus proving noninterference for a program which receives a secret input and produces a public output, amounts to showing that the program behaves like a *constant* program. For example, proving noninterference for the program f amounts to showing that it is equivalent to either $\lambda\ x\ .\ \mathit{true}$ or $\lambda\ x\ .\ \mathit{false}$; it is immediately apparent that these functions do not depend on the secret input x. But how can we prove this for *any* arbitrary definition of f?

The program f may have been defined as the simple function $\lambda\ x\ .\ (\mathsf{not}\ \mathit{false})$ or perhaps the more complex function $\lambda\ x\ .\ (\lambda\ y\ .\ \mathsf{snd}\ (x\ ,\ y))\ \mathit{true})$. Observe, however, that both these programs can be normalized to the equivalent function $\lambda\ x\ .\ \mathit{true}$. In general, although terms in the language may be arbitrarily complex, their *normal forms* (such as $\lambda\ x\ .\ \mathit{true}$) are not. They are simpler, thus well-suited for showing noninterference.

The key idea in this paper is to normalize terms, and prove noninterference by simple structural induction on their normal forms. To illustrate this, we prove noninterference for a static IFC calculus, which we shall call λ_{sec}, based on Haskell's `seclib` library by Russo et al.. We present the typing rules and static semantics for λ_{sec} by extending Moggi's *computational metalanguage* [Moggi 1991] (Section 2). We identify normal forms of λ_{sec}, and establish syntactic properties about a normal form's dependency on its input (Section 3). Using these properties, we show that the normal forms of program f are $\lambda\ x\ .\ \mathit{true}$ or $\lambda\ x\ .\ \mathit{false}$—as expected (Section 4).

To prove noninterference for all terms using normal forms, we implement normalization for λ_{sec} using *normalization by*

[*]Both authors contributed equally to this work.

PLAS'19, January 01–03, 2018, New York, NY, USA
2019.

evaluation (NbE) [Berger et al. 1998] and prove that it preserves the static semantics (Section 5). Using normalization, we prove noninterference for program f and further generalize this proof to *all* terms in λ_{sec} (Section 6) —including, for example, a program which operates on both secret and public values such as $\text{Bool}_{\text{L}} \times \text{Bool}_{\text{H}} \rightarrow \text{Bool}_{\text{L}} \times \text{Bool}_{\text{H}}$. Finally, we conclude by discussing related work and future directions (Section 7).

Unlike earlier proofs, our proof shows that noninterference is an inherent property of the normal forms of λ_{sec}. Since the proof is primarily type and syntax-directed, it provides an appealing alternative to typical semantics based proof techniques. All the main theorems in this paper have been mechanized in the proof assistant Agda[1].

2 The λ_{sec} calculus

In this section we present λ_{sec}, a static IFC calculus that we shall use as the basis for our proof of noninterference. It models the pure fragment of the IFC library `seclib`[2] for Haskell, and is an extension of the calculus developed by Russo et al. [2009] with sum types. `seclib` is a lightweight implementation of static IFC which allows programmers to incorporate untrusted third-party code into their applications while ensuring that it does not leak sensitive data. Below, we recall the public interface (API) of `seclib`:

$$\text{data } S\ (\ell :: Lattice)\ a$$

$$\begin{aligned} &return :: a \rightarrow S\ \ell\ a \\ &(\gg=)\ :: S\ \ell\ a \rightarrow (a \rightarrow S\ \ell\ b) \rightarrow S\ \ell\ b \\ &up\ \ :: \ell_{\text{L}} \sqsubseteq \ell_{\text{H}} \Rightarrow \mathsf{S}\ \ell_{\text{L}}\ a \rightarrow \mathsf{S}\ \ell_{\text{H}}\ a \end{aligned}$$

Similar to other IFC libraries in Haskell such as LIO [Stefan et al. 2011] or MAC [Vassena et al. 2018], `seclib`'s security guarantees rely on exposing the API to the programmer while hiding the underlying implementation. Programs written against the API and the safe parts of the language [Terei et al. 2012] are guaranteed to be *secure-by-construction*; the library enforces security statically through types. As an example, suppose that we have the two point security lattice (see [Denning 1976]) $\{\text{L}, \text{H}\}$ where the only disallowed flow is from secret (H) to public (L), denoted $\text{H} \not\sqsubseteq \text{L}$. The following program written using the `seclib` API is well-typed and—intuitively—secure:

$$\begin{aligned} &example :: S\ \text{L}\ \text{Bool} \rightarrow S\ \text{H}\ \text{Bool} \\ &example\ p\ =\ up\ (p \gg= \lambda\ b \rightarrow return\ (not\ b)) \end{aligned}$$

The function *example* negates the Bool that it receives as input and upgrades its security level from public to secret. On the other hand, had the program tried to downgrade the secret input to public—clearly violating the policy of the security lattice—the typechecker would have rejected the program as ill-typed.

[1] https://github.com/carlostome/ni-nbe
[2] https://hackage.haskell.org/package/seclib

The calculus. λ_{sec} is a simply typed λ-calculus (STLC) augmented with base (uninterpreted) type, unit type, product and sum types, and a security monad type for every security level in a set of labels (denoted by Label). The set of labels may be a lattice, but our development only requires it to be a preorder on the relation $\sqsubseteq$. Throughout the rest of this paper, we use the labels ℓ_{L} and ℓ_{H} and refer to them as *public* and *secret*, although they represent levels in an arbitrary security lattice such that $\ell_{\text{H}} \not\sqsubseteq \ell_{\text{L}}$. Figure 1 defines the syntax of terms, types and contexts of λ_{sec}.

$$\begin{aligned} &\mathsf{Label}\ \ell\ ,\ \ell_{\text{H}}\ ,\ \ell_{\text{L}} \\ &\mathsf{Context}\ \Gamma\ \Delta\ \Sigma ::= \emptyset \mid \Gamma\ ,\ x : \tau \\ &\mathsf{Type}\ \tau\ \tau_1\ \tau_2 ::= \tau_1 \Rightarrow \tau_2 \mid \iota \mid () \\ &\qquad\qquad \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \\ &\qquad\qquad \mid \mathsf{S}\ \ell\ \tau \\ &\mathsf{Term}\ t\ s\ u\ ::= x \mid \lambda\ x\ .\ t \mid t\ s \mid () \\ &\qquad\qquad \mid <t\ ,\ s> \mid \mathsf{fst}\ t \mid \mathsf{snd}\ t \\ &\qquad\qquad \mid \mathsf{left}\ t \mid \mathsf{right}\ t \mid \mathsf{case}\ t\ s\ u \\ &\qquad\qquad \mid \mathsf{return}\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ u \mid \mathsf{up}\ t \end{aligned}$$

Figure 1. The λ_{sec} calculus.

In addition to the standard introduction and elimination constructs for unit, products and sums in STLC, λ_{sec} uses the constructs return, let and up for the security monad $\mathsf{S}\ \ell\ \tau$ which mirrors S from `seclib`. Note that our presentation favours let, as in Moggi [1989], over the Haskell bind ($\gg=$), although both presentations are equivalent—i.e. $t \gg= \lambda\ x\ .\ u$ can be encoded as $\mathsf{let}\ x = t\ \mathsf{in}\ u$.

The typing rules for return and let, shown in Figure 2, ensure that computations over labeled values in the security monad $\mathsf{S}\ \ell\ \tau$ do not leak sensitive data. The construct return allows the programmer to tag a value of type τ with sensitivity label ℓ; and bind enforces that sequences of computations over labeled values stay at the same security level.

$$\boxed{\Gamma \vdash t : \tau}$$

$$\textsc{Return}\ \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{return}\ t : \mathsf{S}\ \ell\ \tau}$$

$$\textsc{Let}\ \frac{\Gamma \vdash t : \mathsf{S}\ \ell\ \tau_1 \qquad \Gamma\ ,\ x : \tau_1 \vdash s : \mathsf{S}\ \ell\ \tau_2}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ s : \mathsf{S}\ \ell\ \tau_2}$$

$$\textsc{Up}\ \frac{\Gamma \vdash t : \mathsf{S}\ \ell_{\text{L}}\ \tau \qquad \ell_{\text{L}} \sqsubseteq \ell_{\text{H}}}{\Gamma \vdash \mathsf{up}\ t : \mathsf{S}\ \ell_{\text{H}}\ \tau}$$

Figure 2. Type system of λ_{sec} (excerpts).

Further, the calculus models the *up* combinator in `seclib` as the construct up. Its purpose is to relabel computations to higher security levels. The rule [Up], shown in Figure 2, statically enforces that information can only flow from ℓ_L to ℓ_H in agreement with the security policy $\ell_L \sqsubseteq \ell_H$. The rest of the typing rules for λ_{sec} are standard [Pierce 2002], and thus omitted here. For a full account we refer the reader to our Agda formalization.

For completeness, the function *example* from earlier can be encoded in the λ_{sec} calculus as follows:[3]

$$example \;=\; \lambda\ s\ .\ \mathsf{up}\ (\mathsf{let}\ b = s\ \mathsf{in}\ \mathsf{return}\ (not\ b))$$

Static semantics. The static semantics of λ_{sec} is defined as a set of equations relating terms of the same type typed under the same environment. The equations characterize pairs of λ_{sec} terms that are equivalent based on β-reduction, η-expansion and other monadic operations. We present the equations for return and let constructs of the monadic type S (à la Moggi [1991]) in Figure 3, and further extend this with equations for the up primitive in Figure 4. The remaining equations are fairly standard [Lindley 2005], and can be found in the Agda formalization. As customary, we use the notation $t_1\ [x/t_2]$ for capture-avoiding substitution of the term t_2 for variable x in term t_1.

$$\boxed{\Gamma \vdash t_1 \approx t_2 : \tau}$$

β-S
$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma , x : \tau \vdash t_2 : \mathsf{S}\ \ell\ \tau}{\Gamma \vdash \mathsf{let}\ x = (\mathsf{return}\ t_1)\ \mathsf{in}\ t_2 \approx t_2\ [x/t_1] : \mathsf{S}\ \ell\ \tau}$$

η-S
$$\frac{\Gamma \vdash t : \mathsf{S}\ \ell\ \tau}{\Gamma \vdash t \approx \mathsf{let}\ x = t\ \mathsf{in}\ (\mathsf{return}\ x) : \mathsf{S}\ \ell\ \tau}$$

γ-S
$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathsf{S}\ \ell\ \tau_1 \qquad \Gamma , x : \tau_1 \vdash t_2 : \mathsf{S}\ \ell\ \tau_2 \\ \Gamma , x : \tau_1 , y : \tau_2 \vdash t_3 : \mathsf{S}\ \ell\ \tau_3\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ (\mathsf{let}\ y = t_2\ \mathsf{in}\ t_3) \approx \\ \mathsf{let}\ x = (\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2)\ \mathsf{in}\ t_3 : \mathsf{S}\ \ell\ \tau_3\end{array}}$$

Figure 3. Static semantics of λ_{sec} (return and let).

The up primitive induces equations regarding its interaction with itself and other constructs in the security monad. In Figure 4, we make the auxiliary condition of up and the label of return explicit using subscripts for better clarity. These equations can be understood as follows:

- Rule δ_1-S. applying up over let is equivalent to distributing it over the subterms of let
- Rule δ_2-S. applying up on an term labeled as return t is equivalent to relabeling t with the final label
- Rule δ_{trans}-S. applying up twice is equivalent to applying it once using the transitivity of the relation $\sqsubseteq$
- Rule δ_{refl}-S. applying up using the reflexive relation $\ell \sqsubseteq \ell$ is equivalent to not applying it

$$\boxed{\Gamma \vdash t_1 \approx t_2 : \tau}$$

δ_1-S
$$\frac{\begin{array}{c}\Gamma \vdash t : \mathsf{S}\ \ell_L\ \tau_1 \\ \Gamma , x : \tau_1 \vdash u : \mathsf{S}\ \ell_L\ \tau_2 \qquad p : \ell_L \sqsubseteq \ell_H\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{up}_p\ (\mathsf{let}\ x = t\ \mathsf{in}\ u) \approx \\ \mathsf{let}\ x = (\mathsf{up}_p\ t)\ \mathsf{in}\ (\mathsf{up}_p\ u) : \mathsf{S}\ \ell_H\ \tau\end{array}}$$

δ_2-S
$$\frac{\Gamma \vdash t : \tau \qquad p : \ell_L \sqsubseteq \ell_H}{\Gamma \vdash \mathsf{up}_p\ (\mathsf{return}_{\ell_L}\ t) \approx \mathsf{return}_{\ell_H}\ t : \mathsf{S}\ \ell_H\ \tau}$$

δ_{TRANS}-S
$$\frac{\begin{array}{c}\Gamma \vdash t : \mathsf{S}\ \ell_L\ \tau \\ p : \ell_L \sqsubseteq \ell_M \qquad q : \ell_M \sqsubseteq \ell_H \qquad r : \ell_L \sqsubseteq \ell_H\end{array}}{\Gamma \vdash \mathsf{up}_q\ (\mathsf{up}_p\ t) \approx \mathsf{up}_r\ t : \mathsf{S}\ \ell_H\ \tau}$$

δ_{REFL}-S
$$\frac{\Gamma \vdash t : \mathsf{S}\ \ell\ \tau \qquad p : \ell \sqsubseteq \ell}{\Gamma \vdash \mathsf{up}_p\ t \approx t : \mathsf{S}\ \ell\ \tau}$$

Figure 4. Static semantics of λ_{sec} (up).

3 Normal forms of λ_{sec}

As discussed in Section 1, our proof of noninterference utilizes syntactic properties of normal forms, and hence relies on normalizing terms in the language to normal forms. Normal forms form a restricted subset of terms in the λ_{sec} calculus which intuitively corresponds to terms that cannot be normalized further. The syntax of normal forms is defined using two well-typed interdependent syntactic categories: *neutral* forms as $\Gamma \vdash_{ne} t : \tau$ (Figure 5) and normal forms as $\Gamma \vdash_{nf} t : \tau$ (Figure 6). Neutral forms are a special case of normal forms which depend entirely on the typing context (e.g., a variable).

Since the definition of neutral and normal forms are merely a syntactic restriction over terms, they can be embedded back into terms of λ_{sec} using a *quotation* function $\ulcorner n \urcorner$. This embedding can be implemented for neutrals and normal forms by simply mapping them to their term-counterparts.

Neutral forms. The neutral forms are terms which are characterized by a property called *neutrality*, which is stated as follows:

Property 3.1 (Neutrality). For a given neutral form of type $\Gamma \vdash_{ne} \tau$, neutrality states that the type τ must occur as a *subformula* of a type in the context Γ.

For example, given a neutral form $\Gamma \vdash_{ne} n : \mathsf{Bool}$, neutrality states that the type Bool must occur as a subformula

[3] In λ_{sec}, the type Bool is encoded as $() + ()$ with $false = \mathsf{left}\ ()$ and $true = \mathsf{right}\ ()$.

$$\boxed{\Gamma \vdash_{ne} t : \tau}$$

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash_{ne} x : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash_{ne} t : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash_{nf} s : \tau_1}{\Gamma \vdash_{ne} t\ s : \tau_2}$$

$$\textsc{Fst}\ \frac{\Gamma \vdash_{ne} t : \tau_1 \times \tau_2}{\Gamma \vdash_{ne} \mathsf{fst}\ t : \tau_1} \qquad \textsc{Snd}\ \frac{\Gamma \vdash_{ne} t : \tau_1 \times \tau_2}{\Gamma \vdash_{ne} \mathsf{snd}\ t : \tau_2}$$

Figure 5. Neutral forms.

$$\boxed{\Gamma \vdash_{nf} t : \tau}$$

$$\textsc{Unit}\ \frac{}{\Gamma \vdash_{nf} () : ()} \qquad \textsc{Lam}\ \frac{\Gamma , x : \tau_1 \vdash_{nf} t : \tau_2}{\Gamma \vdash_{nf} \lambda\, x\, .\, t : \tau_1 \Rightarrow \tau_2}$$

$$\textsc{Base}\ \frac{\Gamma \vdash_{ne} t : \iota}{\Gamma \vdash_{nf} t : \iota} \qquad \textsc{Ret}\ \frac{\Gamma \vdash_{nf} t : \tau}{\Gamma \vdash_{nf} \mathsf{return}\ t : \mathsf{S}\ \ell\ \tau}$$

$$\textsc{LetUp}\ \frac{\ell_L \sqsubseteq \ell_H \qquad \Gamma \vdash_{ne} t : \mathsf{S}\ \ell_L\ \tau_1 \qquad \Gamma , x : \tau_1 \vdash_{nf} s : \mathsf{S}\ \ell_H\ \tau_2}{\Gamma \vdash_{nf} \mathsf{let}{\uparrow}\ x = t\ \mathsf{in}\ s : \mathsf{S}\ \ell_H\ \tau_2}$$

$$\textsc{Left}\ \frac{\Gamma \vdash_{nf} t : \tau_1}{\Gamma \vdash_{nf} \mathsf{left}\ t : \tau_1 + \tau_2} \qquad \textsc{Right}\ \frac{\Gamma \vdash_{nf} t : \tau_2}{\Gamma \vdash_{nf} \mathsf{right}\ t : \tau_1 + \tau_2}$$

$$\textsc{Case}\ \frac{\Gamma \vdash_{ne} t : \tau_1 + \tau_2 \qquad \Gamma , x_1 : \tau_1 \vdash_{nf} t_1 : \tau \qquad \Gamma , x_2 : \tau_2 \vdash_{nf} t_2 : \tau}{\Gamma \vdash_{nf} \mathsf{case}\ t\ (\mathsf{left}\ x_1 \longrightarrow t_1)\ (\mathsf{right}\ x_2 \longrightarrow t_2) : \tau}$$

Figure 6. Normal forms.

of some type in the typing context Γ. An example of such a context is $\Gamma = [x : () \Rightarrow \mathsf{Bool}\ , y : \mathsf{S}\ \ell_H\ \iota]$. The notion of a subformula, originally defined for logical propositional formulas in proof theory [Troelstra and Schwichtenberg 2000], can also be defined for types as follows:

Definition 3.1 (Subformula)**.** For some types τ, τ_1 and τ_2; a subformula of a type is defined as:

- τ is a subformula of τ
- τ is a subformula of $\tau_1 \otimes \tau_2$ if τ is a subformula of τ_1 or τ is a subformula of τ_2, where $\otimes$ denotes the binary type operators $\times$, $+$ and $\Rightarrow$.

The type Bool occurs as a subformula in the typing context $[() \Rightarrow \mathsf{Bool}\ , \mathsf{S}\ \ell_H\ \iota]$ since the type Bool is a subformula of the type $() \Rightarrow \mathsf{Bool}$. Note, however, that the type ι does not occur as a subformula in this context since ι is not a subformula of the type $\mathsf{S}\ \ell_H\ \iota$ by the above definition.

Normal forms. The normal forms are terms characterized by their reducibility: a normal form of type $\Gamma \vdash_{nf} \tau$ cannot be reduced further using the static semantics. Precisely, normal forms are terms which cannot be re-written using the equations in the definition of the relation $\approx$ from left to right. The normal forms in Figure 6 extend the β-short η-long forms in STLC [Balat et al. 2004] [Abel and Sattler 2019] with return and let↑. Note that, unlike neutrals, arbitrary normal forms do not obey neutrality since they may also construct values which do not occur in the context. For example, the normal form left () (which denotes the value *false*) of type $\emptyset \vdash_{nf} \mathsf{Bool}$ constructs a value of the type Bool in the empty context $\emptyset$.

The reader may have noticed that the let↑ construct in normal forms does not directly resemble a term, and hence it is not immediately obvious how it should be quoted. Normal forms constructed by let↑ can be quoted by first applying up to the quotation of the neutral and then using let. The reason let↑ represents both let and up in the normal forms is to retain the non-reducibility of normal forms. Had we added up separately to normal forms, then this may trigger further reductions. For example, the term up (return ()) can be reduced further to the term return (). Disallowing up-terms directly in normal forms disallows the possibility of this reduction in normal forms. Similarly, adding up to neutral forms is also equally worse since it breaks neutrality.

The syntactic characterization of neutral and normal forms provides us with useful properties in the proof of noninterference. For example, there cannot exist a neutral of type $\emptyset \vdash_{ne} \tau$ for any type τ. By neutrality, if such a neutral form exists, then τ must be a subformula of the empty context $\emptyset$, but this is impossible! Similarly, the η-long form of normal forms guarantee that a normal form of a function type must begin with either a λ or case—hence reducing the number of possible cases in our proof. In the next section, we utilize these properties to prove show that the program f (from earlier) behaves as a constant.

4 Normal Forms and Noninterference

The program $f : \mathsf{Int}_H \rightarrow \mathsf{Bool}_L$ from Section 1 can be generalized in λ_{sec} as a term[4] $\emptyset \vdash f : \mathsf{S}\ \ell_H\ \tau \Rightarrow \mathsf{S}\ \ell_L\ \mathsf{Bool}$ marking the secret input and public output through the security monad. Noninterference for this term—which Russo et al. [2009] refer to as a "noninterference-like" property for λ_{sec}—states that given two levels ℓ_L (*public*) and ℓ_H (*secret*) such that the flow $\ell_H \sqsubseteq \ell_L$ is disallowed; for any two possibly different secrets s_1 and s_2, applying f to s_1 is equivalent to applying it to s_2. In other words, it states that varying the secret input must *not interfere* with the public output.

[4] λ_{sec} does not have polymorphic types, in this case τ represents an arbitrary but concrete type, for instance unit ().

As explained earlier, for $\emptyset \vdash f : \mathsf{S}\ \ell_H\ \tau \Rightarrow \mathsf{S}\ \ell_L\ \mathsf{Bool}$ to satisfy noninterference, it must be equivalent to the constant function whose body is return *true* or return *false* independently of the input. For an arbitrary program f it is not possible to conclude so just from case analysis—as programs may be fairly complex—however, for normal forms of the same type it is possible. In the lemma below, we materialize this intuition:

Lemma 4.1 (Normal forms of f are constant)**.** For any normal form $\emptyset \vdash_{nf} f : \mathsf{S}\ \ell_H\ \tau \Rightarrow \mathsf{S}\ \ell_L\ \mathsf{Bool}$, either $f \equiv \lambda\ x\ .\ (\mathsf{return}\ true)$ or $f \equiv \lambda\ x\ .\ (\mathsf{return}\ false)$

The proof follows by direct case analysis on the normal forms of type $\emptyset \vdash_{nf} f : \mathsf{S}\ \ell_H\ \tau \Rightarrow \mathsf{S}\ \ell_L\ \mathsf{Bool}$:

Proof of Lemma 4.1. Upon closer inspection of the normal forms of λ_{sec} (Figure 6), the reader may notice that for the function type $\emptyset \vdash_{nf} \mathsf{S}\ \ell_H\ \tau \Rightarrow \mathsf{S}\ \ell_L\ \mathsf{Bool}$ there exists only two possibilities: a case or a λ construct. The former, can be easily dismissed by neutrality because it requires the scrutinee—a neutral form of sum type $\tau_1 + \tau_2$—to appear in the empty context. In the latter case, the λ construct extends typing context of the body with the type of the argument, and thus refines the normal form to have the shape $\lambda\ x\ .\ \_$ where $\emptyset\ , x : \mathsf{S}\ \ell_H\ \tau \vdash_{nf} \_ : \mathsf{S}\ \ell_L\ \mathsf{Bool}$.

Considering the normal forms of type $\emptyset\ , x : \mathsf{S}\ \ell_H\ \tau \vdash_{nf} \mathsf{S}\ \ell_L\ \mathsf{Bool}$, we realize that there are only three possible candidates: the case construct again, the monadic return or let. As before, case is discharged because it requires the scrutinee of sum type to occur in the context $\emptyset\ , x : \mathsf{S}\ \ell_H\ \tau$. Analogously, the monadic let with a neutral term of type $\mathsf{S}\ \ell_L\ \tau$, expects this type to occur in the same context—but it does not, since $\mathsf{S}\ \ell_L\ \tau$ is not a subformula of $\mathsf{S}\ \ell_H\ \tau$. The remaining case, return, can be further refined, where the only possibilities leave us with $\lambda\ x\ .\ (\mathsf{return}\ true)$ or $\lambda\ x\ .\ (\mathsf{return}\ false)$. □

In order to show that noninterference holds for arbitrary programs of type $\emptyset \vdash f : \mathsf{S}\ \ell_H\ \tau \Rightarrow \mathsf{S}\ \ell_L\ \mathsf{Bool}$ using this lemma, we must link the behaviour of a program with that of its normal form. In the next section we develop the necessary normalization machinery and later complete the proof of noninterference in Section 6.

5 From λ_{sec} to Normal forms

The goal of this section is to implement a normalization algorithm that bridges the gap between terms and their normal forms. For this purpose, we employ Normalization by Evaluation (NbE).

Normalization based on rewriting techniques [Pierce 2002] perform syntactic transformations of a term to produce a normal form. NbE, on the other hand, normalizes a term by evaluating it in a host language, and then extracting a normal form from the (semantic) value in the host language. Evaluation of a term is implemented by an interpreter function eval, and the extraction of normal forms, called *reification*, is implemented by an inverse function reify. Normalization is implemented as a function from terms to normal forms by composing these functions:

$$\begin{aligned} &\mathsf{norm} : (\Gamma \vdash \tau) \to (\Gamma \vdash_{nf} \tau) \\ &\mathsf{norm}\ t = \mathsf{reify}\ (\mathsf{eval}\ t) \end{aligned}$$

The function eval and reify have the following types in the host language:

$$\begin{aligned} &\mathsf{eval} : (\Gamma \vdash \tau) \to (⟦ \Gamma ⟧ \to ⟦ \tau ⟧) \\ &\mathsf{reify} : (⟦ \Gamma ⟧ \to ⟦ \tau ⟧) \to (\Gamma \vdash_{nf} \tau) \end{aligned}$$

In these types, the function $⟦ \_ ⟧$ interprets types and contexts in λ_{sec} as types in the host language. That is, the type $⟦ \tau ⟧$ the denotes the interpretation of the (λ_{sec}) type τ in the host language, and similarly for $⟦ \Gamma ⟧$. On the other hand, the function $⟦ \Gamma ⟧ \to ⟦ \tau ⟧$—a function between the interpretations in the host language—denotes the interpretation of the term $\Gamma \vdash \tau$.

The advantages of using NbE over a rewrite system are two-fold: first, it serves as an actual implementation of the normalization algorithm; second, and the most important advantage, when implemented in a proof system like Agda, it makes normalization amenable to formal reasoning. For example, since Agda ensures that all functions are total, we are assured that a normal form must exist for every term in λ_{sec}. Similarly, we also get a proof that normalization terminates for free since Agda ensures that all functions are terminating.

We implement the functions eval and reify for terms in λ_{sec} using Agda as the host language. Note that, however, the implementation of our algorithm—and NbE in general—is not specific to Agda. It may also be implemented in other programming languages such as Haskell [Danvy et al. 2001] or Standard ML [Balat et al. 2004; Lindley 2005].

In the remainder of this section, we will denote the typing derivations $\Gamma \vdash_{nf} \tau$ and $\Gamma \vdash_{ne} \tau$ as $\mathsf{Nf}\ \tau$ and $\mathsf{Ne}\ \tau$ respectively. We leave the context Γ implicit to avoid the clutter caused by contexts and their *weakenings* [Altenkirch et al. 1995; McBride 2018]. Similarly, we will represent variables of type $\tau \in \Gamma$ as $\mathsf{Var}\ \tau$, leaving Γ implicit. Although we use de Bruijn indices in the actual implementation of variables, we will continue to use named variables here to ease presentation. We encourage the curious reader to see the formalization in Agda for further details.

5.1 NbE for simple types

To begin with, we implement evaluation and reification for the types ι, $()$, $\times$ and $\Rightarrow$. The implementation for sums is a bit more technical, and hence deferred to the Appendix. Note that the implementation of NbE for simple types is entirely standard [Altenkirch et al. 1995; Balat et al. 2004]. Their interpretation as Agda types is defined as follows:

$$\begin{aligned} ⟦ \iota ⟧ &= \mathsf{Nf}\ \iota \\ ⟦ () ⟧ &= \top \end{aligned}$$

$$
\begin{aligned}
[\![\tau_1 \times \tau_2]\!] &= [\![\tau_1]\!] \times [\![\tau_2]\!] \\
[\![\tau_1 \Rightarrow \tau_2]\!] &= [\![\tau_1]\!] \to [\![\tau_2]\!]
\end{aligned}
$$

The types $()$, $\times$ and $\Rightarrow$ are simply interpreted as their counterparts in Agda. For the base type ι, however, we cannot provide a counterpart in Agda since we do not know anything about this type. Instead, since the type ι is not constructed or eliminated by any specific construct in λ_{sec}, we simply require a normal form as an evidence for producing a value of type ι—and thus interpret it as $\mathsf{Nf}\ \iota$.

Typing contexts map variables to types, and hence their interpretation is an execution environment (or equivalently, a semantic substitution) defined like-wise:

$$
\begin{aligned}
[\![\emptyset]\!] &= \emptyset \\
[\![\Gamma , x : \tau_1]\!] &= [\![\Gamma]\!] \, [\, \mathsf{Var}\ \tau_1 \mapsto [\![\tau_1]\!] \,]
\end{aligned}
$$

For example, a value γ which inhabits the interpretation $[\![\Gamma]\!]$ denotes the execution environment for evaluating a term typed in the context Γ.

Given these definitions, evaluation is implemented as a straight-forward interpreter function as follows:

$$
\begin{aligned}
&\mathsf{eval}\ x && \gamma = \mathsf{lookup}\ x\ \gamma \\
&\mathsf{eval}\ () && \gamma = \mathsf{tt} \\
&\mathsf{eval}\ (\mathsf{fst}\ t) && \gamma = \pi_1\ (\mathsf{eval}\ t\ \gamma) \\
&\mathsf{eval}\ (\mathsf{snd}\ t) && \gamma = \pi_2\ (\mathsf{eval}\ t\ \gamma) \\
&\mathsf{eval}\ (< t_1 , t_2 >) && \gamma = (\mathsf{eval}\ t_1\ \gamma , \mathsf{eval}\ t_2\ \gamma) \\
&\mathsf{eval}\ (\lambda\ x\ .\ t) && \gamma = \lambda\ v \to \mathsf{eval}\ t\ (\gamma\ [x \mapsto v]) \\
&\mathsf{eval}\ (t\ s) && \gamma = (\mathsf{eval}\ t\ \gamma)\ (\mathsf{eval}\ s\ \gamma)
\end{aligned}
$$

Note that γ is an execution environment for the term's context; lookup, π_1 and π_2 are Agda functions; and tt is the constructor of the unit type $\top$. For the case of $\lambda\ x\ .\ t$, evaluation is expected to return an equivalent semantic function. We compute the body of this function by evaluating the body term t using the substitution γ extended with a mapping which assigns the value v to the variable x—denoted $\gamma\ [x \mapsto v]$.

Reification, on the other hand, is implemented using two helper functions reflect and reifyVal. The function reflect converts neutral forms to semantic values, while the dual function reifyVal converts semantic values to normal forms. These functions are implemented as follows:

$$
\begin{aligned}
&\mathsf{reifyVal} : [\![\tau]\!] \to \mathsf{Nf}\ \tau \\
&\mathsf{reifyVal}\ \{\iota\}\ n = n \\
&\mathsf{reifyVal}\ \{()\}\ \mathsf{tt} = () \\
&\mathsf{reifyVal}\ \{\tau_1 \times \tau_2\}\ p = \\
&\quad < \mathsf{reifyVal}\ \{\tau_1\}\ (\pi_1\ p) , \mathsf{reifyVal}\ \{\tau_2\}\ (\pi_1\ p) > \\
&\mathsf{reifyVal}\ \{\tau_1 \Rightarrow \tau_2\}\ f = \\
&\quad \lambda\ x\ .\ \mathsf{reifyVal}\ \{\tau_2\}\ (f\ (\mathsf{reflect}\ \{\tau_1\}\ x)) \mid \mathit{fresh}\ x
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{reflect} : \mathsf{Ne}\ \tau \to [\![\tau]\!] \\
&\mathsf{reflect}\ \{\iota\}\ n = n \\
&\mathsf{reflect}\ \{()\}\ n = \mathsf{tt} \\
&\mathsf{reflect}\ \{\tau_1 \times \tau_2\}\ n = \\
&\quad (\mathsf{reflect}\ \{\tau_1\}\ (\mathsf{fst}\ n) , \mathsf{reflect}\ \{\tau_2\}\ (\mathsf{snd}\ n)) \\
&\mathsf{reflect}\ \{\tau_1 \Rightarrow \tau_2\}\ n = \\
&\quad \lambda\ v \to \mathsf{reflect}\ \{\tau_2\}\ (n\ (\mathsf{reifyVal}\ \{\tau_1\}\ v))
\end{aligned}
$$

Note that the argument inside the braces { } denotes an implicit parameter, which is the type of the corresponding neutral/value argument of reflect/reifyVal here.

Reflection is implemented by performing a type-directed translation of neutral forms to semantic values by induction on types. The interpretation of types, defined earlier, guides our implementation. For example, reflection of a neutral with a function type must produce a function value since the type $\Rightarrow$ is interpreted as an Agda function. For this purpose, we are given the argument value in the semantics and it remains to construct a function body of the appropriate type. We produce the body of this function by recursively reflecting a neutral application of the function and (the reification of) the argument value. The function reifyVal is also implemented in a similar fashion by induction on types.

To implement reification, recollect that the argument to reify is a function that results from partially applying the eval function with a term. If the term has type $\Gamma \vdash \tau$, then the argument, say f, must have the type $[\![\Gamma]\!] \to [\![\tau]\!]$. Thus, to apply f, we need an execution environment of the type $[\![\Gamma]\!]$. This environment can be generated by simply reflecting the variables in the context as follows:

$$
\begin{aligned}
&\mathsf{genEnv} : (\Gamma : \mathsf{Ctx}) \to [\![\Gamma]\!] \\
&\mathsf{genEnv}\ \emptyset = \emptyset \\
&\mathsf{genEnv}\ (\Gamma , x : \tau) = \mathsf{genEnv}\ \Gamma\ [\, x \mapsto \mathsf{reflect}\ x \,]
\end{aligned}
$$

Finally, we can now implement reify as follows:

$$
\mathsf{reify}\ \{\Gamma\}\ f = \mathsf{let}\ \gamma = \mathsf{genEnv}\ \Gamma\ \mathsf{in}\ \mathsf{reifyVal}\ (f\ \gamma)
$$

We generate an environment γ to apply the semantic function f, and then convert the resulting semantic value to a normal form by applying reifyVal.

5.2 NbE for the security monad

To interpret a type $\mathsf{S}\ \ell\ \tau$, we need a semantic counterpart in the host language which is also a monad. Suppose that we define such a monad as an inductive data type T parameterized by a label ℓ and some type a (which would be $[\![\tau]\!]$ in this case). Evidently this monad must allow the implementation of the semantic counterparts of the terms return, let and up in λ_{sec} as follows:

$$
\begin{aligned}
&\mathsf{return} : a \to T\ \ell\ a \\
&\mathsf{bind} \quad : T\ \ell\ a \to (a \to T\ \ell\ b) \to T\ \ell\ b \\
&\mathsf{up} \quad\ \ : (\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}) \to T\ \ell_{\mathrm{L}}\ a \to T\ \ell_{\mathrm{H}}\ a
\end{aligned}
$$

To satisfy this specification, we define the data type T in Agda with the following constructors:

$$\dfrac{x : a}{\mathsf{return}\ x : T\ \ell\ a}\ \textsc{Return}$$

$$\dfrac{p : \ell_L \sqsubseteq \ell_H \qquad n : \mathsf{Ne}\ \mathsf{S}\ \ell_L\ \tau \qquad f : \mathsf{Var}\ \tau \rightarrow T\ \ell_H\ a}{\mathsf{bindNe}\ p\ n\ f : T\ \ell_H\ a}\ \textsc{BindN}$$

The constructor return returns a semantic value in the monad, while bindNe registers a binding of a neutral to monadic value. These constructors are the semantic equivalent of return and let↑ in the normal forms, respectively. The constructor bindNe is more general than the required function bind in order to allow the definition of up, which is defined by induction as follows:

$$\begin{aligned}
&\mathsf{up}\ p\ (\mathsf{return}\ v)\ =\\
&\quad \mathsf{return}\ v\\
&\mathsf{up}\ p\ (\mathsf{bindNe}\ q\ n\ f)\ =\\
&\quad \mathsf{bindNe}\ (\mathsf{trans}\ q\ p)\ n\ (\lambda\ x \rightarrow \mathsf{up}\ p\ (f\ x))
\end{aligned}$$

To understand this implementation, suppose that $p : \ell_M \sqsubseteq \ell_H$ for some labels ℓ_M and ℓ_H. A monadic value of type $T\ \ell_M\ a$ which is constructed by a return can be simply re-labeled to $T\ \ell_H\ a$ since return can be used to construct a monadic value on any label. For the case of bindNe $q\ n\ f$, we have that $q : \ell_L \sqsubseteq \ell_M$ and $n : \mathsf{Ne}\ \mathsf{S}\ \ell_L\ \tau_1$, hence $\ell_L \sqsubseteq \ell_H$ by transitivity, and we may simply use bindNe to register n and recursively apply up on the continuation f to produce the desired result of type $T\ \ell_H\ a$.

Using the type T in the host language, we may now interpret the monad in λ_{sec} as follows:

$$[\![\ \mathsf{S}\ \ell\ \tau\ ]\!]\ =\ T\ \ell\ [\![\ \tau\ ]\!]$$

Having mirrored the monadic primitives in λ_{sec} using semantic counterparts, evaluation is rather simple:

$$\begin{aligned}
&\mathsf{eval}\ (\mathsf{return}\ t) \qquad \gamma\ =\ \mathsf{return}\ (\mathsf{eval}\ t\ \gamma)\\
&\mathsf{eval}\ (\mathsf{up}\ p\ t) \qquad \gamma\ =\ \mathsf{up}\ p\ (\mathsf{eval}\ t\ \gamma)\\
&\mathsf{eval}\ (\mathsf{let}\ x\ =\ t\ \mathsf{in}\ s)\ \gamma\ =\\
&\quad \mathsf{bind}\ (\mathsf{eval}\ t\ \gamma)\ (\lambda\ v \rightarrow \mathsf{eval}\ s\ (\gamma\ [x \mapsto v]))
\end{aligned}$$

For implementing reflection, we can use bindNe to register a neutral binding and recursively reflect the given variable:

$$\begin{aligned}
&\mathsf{reflect}\ \{\mathsf{S}\ \ell\ \tau\}\ n\ =\\
&\quad \mathsf{bindNe}\ \mathsf{refl}\ n\ (\lambda\ x \rightarrow \mathsf{reflect}\ \{\tau\}\ x)
\end{aligned}$$

Since we do not need to increase the sensitivity of the neutral to bind it here, we simply provide the "reflexive flow" $\mathsf{refl} : \ell \sqsubseteq \ell$.

The function reifyVal, on the other hand, is rather straightforward since the constructors of T are essentially semantic counterparts of the normal forms, and can hence be translated to it:

$$\begin{aligned}
&\mathsf{reifyVal}\ \{\mathsf{S}\ \ell\ \tau\}\ (\mathsf{return}\ v)\ =\\
&\quad \mathsf{return}\ (\mathsf{reifyVal}\ \{\tau\}\ v)\\
&\mathsf{reifyVal}\ \{\mathsf{S}\ \ell\ \tau\}\ (\mathsf{bindNe}\ \{p\}\ n\ f)\ =\\
&\quad \mathsf{let}{\uparrow}\ \{p\}\ x\ =\ n\ \mathsf{in}\ \mathsf{reifyVal}\ \{\tau\}\ (f\ x)
\end{aligned}$$

5.3 Preservation of semantics

To prove that normalization preserves static semantics of λ_{sec}, we must show that the normal form of term is equivalent to the term. Since normal forms and terms belong to different syntactic categories, we must first quote normal forms to state this relationship using the term equivalence relation $\approx$. This property, called *consistency* of normal forms, is stated as follows:

Theorem 5.1 (Consistency of normal forms)**.** For any term $\Gamma \vdash t : \tau$ we have that $\Gamma \vdash t \approx \ulcorner\ \mathsf{norm}\ t\ \urcorner : \tau$

An attempt to prove consistency by induction on the terms or types fails quickly since the induction principle alone is not strong enough to prove this theorem. To solve this issue we must establish a notion of equivalence between a term and its interpretation using *logical relations* [Plotkin 1973]. Using these relations, we can prove that evaluation is consistent by showing that it is *related* to applying a substitution in the syntax. Following this, we can also prove the consistency of reification by showing that reifying a value related to a term, yields a normal form which is equivalent to the term when quoted. The consistency of evaluation and reification yields the proof of consistency for normal forms.

This proof follows the style of the consistency proof of NbE for STLC using Kripke logical relations by Coquand [1993]. As is the case for sums, NbE for the security monad uses an inductively defined data type to implement the semantic monad. Hence, we are able to leverage the proof techniques used to prove the consistency of NbE for sums [Valliappan and Russo 2019] to prove the same for the security monad. We skip the details of the proof here, but encourage the curious reader to see the mostly complete[5] Agda formalization of this theorem.

6 Noninterference for λ_{sec}

After developing the necessary machinery to normalize terms in the calculus, we are ready to state and prove noninterference for λ_{sec}. First, we complete the proof of noninterference for the program f from Section 4.

6.1 Special Case of Noninterference

Theorem 6.1 (Noninterference for f)**.** Given security levels ℓ_L and ℓ_H such that $\ell_H \not\sqsubseteq \ell_L$ and a function $\emptyset \vdash f : \mathsf{S}\ \ell_H\ \tau \Rightarrow \mathsf{S}\ \ell_L\ \mathsf{Bool}$ then $\forall\ s_1\ s_2 : \mathsf{S}\ \ell_H\ \tau.\ f\ s_1 \approx f\ s_2$

[5] Although we have proved consistency for the security monad extensions, the complete proof—without open goals for standard results—is still work in progress

The proof of Theorem 6.1 relies upon two key ingredients; Lemma 4.1 (Section 4), which characterizes the shape of the normal forms of f; and consistency of normal forms, Theorem 5.1 (Section 5.3), which links the semantics of f with that of its normal forms.

Proof of Theorem 6.1. To show that a function $\Gamma \vdash f : \mathsf{S}\ \ell_H\ \tau \Rightarrow \mathsf{S}\ \ell_L\ \mathsf{Bool}$ is equivalent when applied to two different secret inputs s_1 and s_2, first, we instantiate Lemma 4.1 with the normal form of f, denoted by $\mathsf{norm}\ f$. In this manner, we obtain that the normal forms of f are exactly the constant function that returns *true* or *false* wrapped in the return. In the former case, by correctness of normalization we have that $f \approx \ulcorner \mathsf{norm}\ f \urcorner \approx \lambda\ x\ .\ \mathsf{return}\ \mathit{true}$. By β-reduction and congruence of term-level function application, we have that $\forall\ t.\ (\lambda\ x\ .\ \mathsf{return}\ \mathit{true})\ t \approx \mathsf{return}\ \mathit{true}$. Therefore, $f\ s_1 \approx f\ s_2$. The case when $\mathsf{norm}\ f \equiv \lambda\ x\ .\ \mathsf{return}\ \mathit{false}$ follows a similar argument. □

The noninterference property proven above characterizes what it means for a concrete class of programs, i.e. those of type $\emptyset \vdash f : \mathsf{S}\ \ell_H\ \tau \Rightarrow \mathsf{S}\ \ell_L\ \mathsf{Bool}$, to be secure: the attacker cannot even learn one bit of the secret from using program f. The rest of this section is dedicated to generalize and prove noninterference from the program f to arbitrary programs written in λ_{sec}. As will become clear, normal forms of λ_{sec} play a crucial role towards proving noninterference.

6.2 General Noninterference theorem

In order to discuss general noninterference for λ_{sec}, we must first specify what the *secret* (ℓ_H) inputs of a program and its *public* (ℓ_L) output are, with respect to an attacker at level ℓ_L. The attacker can only learn information of a program by running it with different secret inputs and then observing its public output. Because the attacker can only observe outputs at their security level, we restrict the security condition to only consider programs where outputs are fully observable, i.e., *transparent* and *ground*, to the attacker.

This restriction does not hinder the generality of our security condition: a program producing a partially public output, for instance a product $\mathsf{S}\ \ell_L\ \mathsf{Bool} \times \mathsf{S}\ \ell_H\ \mathsf{Bool}$, can be transformed to produce a fully public output by applying the snd projection. We return to this example later in the next section. Also note that previous work on proving noninterference for static IFC languages [Abadi et al. 1999; Miyamoto and Igarashi 2004] also impose such a restriction.

First, we define the transparent and ground properties for types:

Definition 6.1 (Transparent type)**.**

- $()$ is transparent at any level ℓ.
- ι is transparent at any level ℓ.
- $\tau_1 \Rightarrow \tau_2$ is transparent at ℓ iff τ_2 is transparent at ℓ.
- $\tau_1 + \tau_2$ is transparent at ℓ iff τ_1 and τ_2 are transparent at ℓ.
- $\tau_1 \times \tau_2$ is transparent at ℓ iff τ_1 and τ_2 are transparent at ℓ.
- $\mathsf{S}\ \ell'\ \tau$ is transparent at ℓ iff $\ell' \sqsubseteq \ell$ and τ is transparent at ℓ.

Definition 6.2 (Ground type)**.**

- $()$ is ground.
- ι is ground.
- $\tau_1 + \tau_2$ is ground iff τ_1 and τ_2 are ground.
- $\tau_1 \times \tau_2$ is ground iff τ_1 and τ_2 are ground.
- $\mathsf{S}\ \ell\ \tau$ is ground iff τ is ground.

A type τ is transparent at security level ℓ_L if the type does not include the security monad type over a higher security level ℓ_H. Further, we restrict the output type to be first order, i.e. types not including the function space, because these are the only values from which the attacker can directly learn information.

Departing from the traditional view of programs as closed terms, i.e. terms without free variables, in the λ_{sec} calculus we consider all terms for which a typing derivation exists as programs. This includes terms that contain free variables—unknowns—typed by the context, which we identify as the program inputs. Note that open terms are more general since they can always be closed as a function by abstracting over the free variables.

Now, we state what it means for a context to appear secret at level ℓ. These definitions, dubbed ℓ-sensitivity, force the types appearing in the context to be at least as sensitive as ℓ.

Definition 6.3 (Type sensitivity)**.**
A type τ is ℓ-sensitive if and only if:

- τ is the function type $\tau_1 \Rightarrow \tau_2$ and τ_2 is ℓ-sensitive.
- τ is the product type $\tau_1 \times \tau_2$ and τ_1 and τ_2 are ℓ-sensitive.
- τ is the monadic type $\mathsf{S}\ \ell'\ \tau_1$ and $\ell \sqsubseteq \ell'$.

Definition 6.4 (Context sensitivity)**.**
A context Γ is ℓ-sensitive if and only if for all types $\tau \in \Gamma$, τ is ℓ-sensitive.

Next, we formally introduce substitutions[6] which lay at the core of β-reduction rule for the λ_{sec} calculus. Substitutions map free variables in a term to other terms possibly typed in a different context.

$$\mathit{Substitution}\ \sigma ::= \sigma_\emptyset \mid \sigma\,[x \mapsto t]$$

$$\boxed{\Gamma \vdash_{sub} \sigma : \Delta}$$

$$\frac{\Gamma \vdash_{sub} \sigma : \Delta \qquad \Gamma \vdash t : \tau}{\Gamma \vdash_{sub} \sigma\,[\,x \mapsto t\,] : \Delta\,,\,x : \tau} \qquad \frac{}{\Gamma \vdash_{sub} \sigma_\emptyset : \emptyset}$$

Figure 7. Substitutions of λ_{sec}

[6]In Section 2 we purposely left capture-avoiding substitutions underspecified, we amend that here.

A substitution is either empty, $\sigma_\emptyset$, or is the substitution σ extended with a new mapping from the variable $x : \tau$ to term t. We denote $t\,[\,\sigma\,]$ the application of substitution σ to term t. Its definition is standard by induction on the term structure thus we omit it here and refer the reader to the Agda formalization.

Substitutions, in general, provide a mix of terms of secret and public type to fill the variables in the context Γ of a program. However, for noninterference we need to fix the public part of the substitution and allow the secret part to vary. We do so by splitting a substitution σ into the composition of a public substitution, $\Gamma \vdash_{\text{sub}} \sigma_{\ell_L} : \Delta$, that fixes the public inputs, and a secret substitution $\Delta \vdash_{\text{sub}} \sigma_{\ell_H} : \Sigma$, that restricts Δ to be ℓ_{H}-sensitive. The composition of both, denoted $\Gamma \vdash_{\text{sub}} (\sigma_{\ell_L}\ ;\ \sigma_{\ell_H}) : \Sigma$, maps variables in context Γ to terms typed in Σ: first, σ_{ℓ_L} maps variables from Γ to terms in Δ, subsequently, σ_{ℓ_H} maps variables in Δ to terms typed in Σ. Below, we state ℓ_{L}-equivalence of substitutions:

Definition 6.5 (Low equivalence of substitutions)**.**
Two substitutions σ_1 and σ_2 are ℓ_{L}-equivalent , written $\sigma_1 \approx_{\ell_L} \sigma_2$, if and only if for all ℓ_{H} such that $\ell_{\text{H}} \not\sqsubseteq \ell_{\text{L}}$, there exists a public substitution σ_{ℓ_L}, and two secret substitutions $\sigma^1_{\ell_H}$ and $\sigma^2_{\ell_H}$, such that $\sigma_1 \equiv \sigma_{\ell_L}\ ;\ \sigma^1_{\ell_H}$ and $\sigma_2 \equiv \sigma_{\ell_L}\ ;\ \sigma^2_{\ell_H}$

Informally, noninterference for λ_{sec} states that applying two low equivalent substitutions to an arbitrary term whose type is ground and transparent yields two equivalent programs. As previously explained, intuitively a program satisfies such property if it is equivalent to a *constant* program: i.e. a program where the output does not depend on the input—in this case the variables in the typing context. As in Section 4, instead of defining and proving this on arbitrary terms, we achieve this using normal forms.

Constant terms and normal forms. We prove the noninterference theorem by showing that terms typed in a ℓ_{H}-sensitive context must be constant. We achieve this in turn by showing that the normal forms of such terms are also constant with respect to the context. Below, we formally state what it means for a term to be constant:

Definition 6.6 (Constant term)**.**
A term $\Gamma \vdash t : \tau$ is said to be constant if, for any two substitutions σ_1 and σ_2, we have that $t\,[\,\sigma_1\,] \approx t\,[\,\sigma_2\,]$.

Similarly, we must define what it means for a normal form to be constant. However, we cannot state this for normal forms directly using substitutions since the result of applying a substitution to a normal form may not be a normal form. For example, the result of substituting the variable x in the normal form $x : \iota \Rightarrow \iota\ ,\ y : \iota \vdash_{\text{nf}} x\ y : \iota$ by the identity function is not a normal form—and *cannot* be derived syntactically as a normal form using $\vdash_{\text{nf}}$. Instead, we lean on the shape of the context to state the property. If a normal form $\Gamma \vdash_{\text{nf}} n : \tau$ is constant, then there must exist a syntactically identical derivation $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $n \equiv n'$. However, since n and n' are typed in different contexts, Γ and $\emptyset$, it is not possible to compare them for syntactic equality as in $n \equiv n'$. We solve this problem by *renaming* the normal form n' to add as many variables as mentioned in context Γ. Renaming is defined as follows:

$$\mathsf{ren} : \{\Gamma \leqslant \Delta\} \to (\Gamma \vdash_{\text{nf}} \tau) \to (\Delta \vdash_{\text{nf}} \tau)$$

The relation $\leqslant$ between contexts Γ and Δ indicates that the variables in Δ are at least those in Γ. We define such relation, called weakening of contexts, below:

Definition 6.7 (Weakening of contexts)**.**
The binary relation $\leqslant$ on contexts is defined as follows:

- $\emptyset \leqslant \emptyset$
- If $\Gamma \leqslant \Delta$, then $\Gamma \leqslant \Delta\ ,\ x : \tau$
- If $\Gamma \leqslant \Delta$, then $\Gamma\ ,\ x : \tau \leqslant \Delta\ ,\ x : \tau$

The function ren is defined by simple structural induction on the derivation of the normal forms. Note that terms can also be renamed in the same fashion by using $\approx$ instead of $\equiv$. Using renaming, we now define when a normal form is constant as follows:

Definition 6.8 (Constant normal form)**.** A normal form $\Gamma \vdash_{\text{nf}} n : \tau$ is constant if there exists a normal form $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $\mathsf{ren}\ (n') \equiv n$.

Further, we need a lemma showing that if a term is constant, then so is its normal form.

Lemma 6.2 (Constant plumbing lemma)**.** If the normal form n of a term $\Gamma \vdash t : \tau$ is constant, then so is t.

The proof follows by induction on the normal forms:

Proof of Lemma 6.2. If n is constant, then there must exist a normal form $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $\mathsf{ren}\ (n') \equiv n$. Let the quotation of this normal form $\ulcorner n' \urcorner$ be some term $\emptyset \vdash t' : \tau$. Recall from earlier that terms can also be renamed, hence we have $\mathsf{ren}\ (t') \approx \mathsf{ren}\ (\ulcorner n' \urcorner)$ by correctness of n'. Since it can be shown that $\mathsf{ren}\ (\ulcorner n' \urcorner) \equiv \ulcorner \mathsf{ren}\ (n') \urcorner$, we have that $\mathsf{ren}\ (\ulcorner n' \urcorner) \equiv \ulcorner n \urcorner$, and by correctness of n, we also have $\mathsf{ren}\ (t') \approx t$ — (1).

A substitution σ maps free variables in a term to terms. The empty substitution, denoted $\sigma_\emptyset$, is the unique substitution, such that $\Delta \vdash t'\,[\,\sigma_\emptyset\,] : \tau$ for any Δ. That is, applying the empty substitution simply renames the term. We can show that $t'\,[\,\sigma_\emptyset\,] \equiv \mathsf{ren}\ (t')$, and hence, by (1), we have $t'\,[\,\sigma_\emptyset\,] \approx t$ — (2). Since $\sigma_\emptyset$ is unique, we can show that for any substitution σ, we have $(t'\,[\,\sigma_\emptyset\,])\,[\,\sigma\,] \approx t'\,[\,\sigma_\emptyset\,]$. From this, for any two substitutions σ_1 and σ_2, we have $(t'\,[\,\sigma_\emptyset\,])\,[\,\sigma_1\,] \approx (t'\,[\,\sigma_\emptyset\,])\,[\,\sigma_2\,]$ by transitivity of $\approx$. As a result, from (2), we achieve the desired result, $t\,[\,\sigma_1\,] \approx t\,[\,\sigma_2\,]$, therefore t must be constant. □

Next, normal forms typed in sensitive contexts are constant or the flow between context and output type is permitted as shown below:

Lemma 6.3 (Normal forms do not leak)**.** Given a normal form $\Gamma \vdash_{\mathrm{nf}} n : \tau$, where the context Γ is ℓ_i-sensitive, and τ is a ground and transparent type at level ℓ_o, then either n is constant or $\ell_i \sqsubseteq \ell_o$.

Proof. By induction on the structure of the normal form n. Note that λ and case normal forms need not be considered since the preconditions ensure that τ cannot be a function type (dismisses λ), and Γ cannot contain a variable of a sum type (dismisses case).

- **Case 1** ($\Gamma \vdash_{\mathrm{nf}} () : ()$). The normal form $()$ is constant.
- **Case 2** ($\Gamma \vdash_{\mathrm{nf}} n : \iota$). In this case, we are given that n is a neutral, by the [Base] rule in Figure 6. It can be shown by simple induction that for all neutrals of type $\Gamma \vdash_{\mathrm{ne}} \tau$, if Γ is ℓ_i-sensitive and τ is transparent at ℓ_o, then $\ell_i \sqsubseteq \ell_o$. Hence, in this case, the neutral n gives us that $\ell_i \sqsubseteq \ell_o$.
- **Case 3** ($\Gamma \vdash_{\mathrm{nf}}$ return $n : \mathsf{S}\ \ell\ \tau$). By applying the induction hypothesis on the normal form n, we have that n is either constant or $\ell_i \sqsubseteq \ell_o$. In the latter case, we are done since we already have $\ell_i \sqsubseteq \ell_o$. In the former case, we have that there exists an normal form n' such that ren $(n') \equiv n$. By congruence of the relation $\equiv$, we get that return (ren (n')) $\equiv$ return n. Note that the function ren is defined as ren (return n') $\equiv$ return (ren n'), and hence by transitivity of $\equiv$, we have that ren (return (n')) $\equiv$ return n. Thus, the normal form return n is also constant.
- **Case 4** ($\Gamma \vdash_{\mathrm{nf}}$ let$\uparrow x = n$ in $m : \mathsf{S}\ \ell_2\ \tau_2$). For this case, we have a neutral $\Gamma \vdash_{\mathrm{ne}} n : \mathsf{S}\ \ell_1\ \tau_1$ such that $\ell_1 \sqsubseteq \ell_2$, by the [LetUp] rule in Figure 6. Similar to case 2, we have that $\ell_i \sqsubseteq \ell_1$ from the neutral n. Hence, $\ell_i \sqsubseteq \ell_2$ by transitivity of the relation $\sqsubseteq$. Additionally, since $\mathsf{S}\ \ell_2\ \tau$ is transparent at ℓ_o, it must be the case that $\ell_2 \sqsubseteq \ell_o$ by definition of transparency. Therefore, once again by transitivity, we have $\ell_i \sqsubseteq \ell_o$.
- **Case 5** ($\Gamma \vdash_{\mathrm{nf}}$ left $n : \tau_1 + \tau_2$). Similar to return.
- **Case 6** ($\Gamma \vdash_{\mathrm{nf}}$ right $n : \tau_1 + \tau_2$). Similar to return.

$\square$

The last step before noninterference is an ancillary lemma which shows that terms typed in ℓ_{H}-sensitive contexts are constant:

Lemma 6.4. Given a term $\Gamma \vdash t : \tau$, where the context Γ is ℓ_{H}-sensitive, and τ is a ground type transparent at ℓ_{L}. If $\ell_{\mathrm{H}} \not\sqsubseteq \ell_{\mathrm{L}}$, then t is constant.

Proof of Lemma 6.4. By Lemma 6.3, we have that the normal form of t is constant since $\ell_{\mathrm{H}} \not\sqsubseteq \ell_{\mathrm{L}}$. Hence, by Lemma 6.2, t must also be constant. $\square$

Finally, we are ready to formally state and prove the noninterference property for programs written in λ_{sec}, effectively showing that programs do not leak sensitive information. The proof follows the previous lemmas, which characterize the behaviour of programs by the syntactic properties of their normal forms.

Theorem 6.5 (Noninterference for λ_{sec})**.** Given security levels ℓ_{L} and ℓ_{H} such that $\ell_{\mathrm{H}} \not\sqsubseteq \ell_{\mathrm{L}}$; an attacker at level ℓ_{L}; two ℓ_{L}-*equivalent* substitutions σ_1 and σ_2 such that $\sigma_1 \approx_{\ell_L} \sigma_2$; and a type τ that is ground and transparent at ℓ_{L}; then for any term $\Gamma \vdash t : \tau$ we have that $t\,[\,\sigma_1\,] \approx t\,[\,\sigma_2\,]$.

Proof of Theorem 6.5. Low equivalence of substitutions $\sigma_1 \approx_{\ell_L} \sigma_2$ gives that $\sigma_1 = \sigma_{\ell_L}\ ;\ \sigma^1_{\ell_H}$ and $\sigma_2 = \sigma_{\ell_L}\ ;\ \sigma^2_{\ell_H}$. After applying the public substitution σ_{ℓ_L} to the term $\Gamma \vdash t : \tau$, we are left with a term typed in a ℓ_{H}-sensitive context Δ, $\Delta \vdash t\,[\,\sigma_{\ell_L}\,] : \tau$. By Lemma 6.4, $t\,[\,\sigma_{\ell_L}\,]$ is constant which means that $(t\,[\,\sigma_{\ell_L}\,])\,[\,\sigma^1_{\ell_H}\,] \approx t\,([\,\sigma_{\ell_L}\,])\,[\,\sigma^2_{\ell_H}\,]$. By readjusting substitutions using composition we obtain $t\,([\,\sigma_{\ell_L}\ ;\ \sigma^1_{\ell_H}\,]) \approx t\,([\,\sigma_{\ell_L}\ ;\ \sigma^2_{\ell_H}\,])$, which yields $t\,[\,\sigma_1\,] \approx t\,[\,\sigma_2\,]$. $\square$

6.3 Follow-up Example

To conclude this section, we briefly show how to instantiate the theorem of noninterference for λ_{sec} for programs of type $\emptyset \vdash t : \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool} \times \mathsf{S}\ \ell_{\mathrm{H}}\ \mathsf{Bool} \Rightarrow \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool} \times \mathsf{S}\ \ell_{\mathrm{H}}\ \mathsf{Bool}$, which are the recurring example for explaining noninterference in the literature [Bowman and Ahmed 2015; Russo et al. 2009]. Adapted to the notion of noninterference based on substitutions, the corollary we aim to prove is the following:

Corollary 6.6 (Noninterference for t)**.** Given security levels ℓ_{L} and ℓ_{H} such that $\ell_{\mathrm{H}} \not\sqsubseteq \ell_{\mathrm{L}}$ and a program $x : \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool} \times \mathsf{S}\ \ell_{\mathrm{H}}\ \mathsf{Bool} \vdash t\ \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool} \times \mathsf{S}\ \ell_{\mathrm{H}}\ \mathsf{Bool}$ then $\forall\, p : \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool}$, $s_1\ s_2 : \mathsf{S}\ \ell_{\mathrm{H}}\ \mathsf{Bool}$. we have that $t\,[\,x \mapsto (p\,,\,s_1)\,] \approx t\,[\,x \mapsto (p\,,\,s_2)\,]$.

Because the main noninterference theorem requires the output to be fully observable by the attacker, we transform t to the desired shape by applying the snd projection. This is justified because the first component of the output is protected at level ℓ_{H}, which the attacker cannot observe. Below we prove noninterference for $x : \mathsf{S}\ \ell_{\mathrm{L}}\ \mathsf{Bool} \times \mathsf{S}\ \ell_{\mathrm{H}}\ \mathsf{Bool} \vdash$ snd $t : \mathsf{S}\ \ell_{\mathrm{H}}\ \mathsf{Bool}$:

Proof of Corollary 6.6. To apply Theorem 6.5 we have to show that both substitutions are low equivalent, $[\,x \mapsto (p\,,\,s_1) \approx_{\ell_L} x \mapsto (p\,,\,s_2)$ The key idea is that the substitution $x \mapsto (p\,,\,s_1)$ can be decomposed into a public substitution $\sigma_{\ell_L} \equiv x \mapsto (p\,,\,y)$ and two different secret substitutions where each replaces variable y by a different secret, $\sigma^1_{\ell_H} \equiv y \mapsto s_1$ and $\sigma^2_{\ell_H} \equiv y \mapsto s_2$. Now, the proof follows directly from Theorem 6.5. $\square$

7 Conclusions and future work

In this paper we have presented a novel proof of noninterference for the λ_{sec} calculus (based on Haskell's IFC library `seclib`) using normalization. The simplicity of the proof relies upon the normal forms of the calculus, which as opposed to arbitrary terms, are well-principled. To normalize terms to normal forms, we have implemented normalization using NbE, and shown that normal forms obey useful syntactic-properties such as neutrality and $\beta\eta$-long form. Most of the auxiliary lemmas and definitions towards proving noninterference build on these properties. Because normal forms are well-principled, in many instances the proofs follow directly by simple structural induction them.

An important difference between our work and previous proofs based on term erasure is that our proof strictly utilizes the static semantics of the language instead of the dynamic semantics. Specifically, our proof of noninterference is not tied to any particular evaluation strategy, such as call-by-name or call-by-value, assuming the strategy is adequate with respect to the static semantics.

Perhaps the closest to our line of work is the proof of noninterference by Miyamoto and Igarashi [2004] for a modal lambda calculus using normalization. The main novelty of our proof is that it works for standard extensions of the simply typed lambda calculus and does not change the typing rules of the underlying calculus (as presented and implemented by Russo et al. [2009]). This makes our proof technique applicable even in the presence of other useful normalization-preserving extensions of STLC. For example, it should be possible to extend our proof for λ_{sec} further with exceptions and other *computational* effects (à la Moggi [1989]) since our security monad is already an instance of this. Moreover, our proof relies on syntactic properties of normal forms in an open typing context since normalization is based on the static semantics of the language. In this work we have only considered the terminating fragment of λ_{sec}, which enabled us to implement the normalization function in a total language such as Agda. It remains an open question whether we could adapt our work to languages which implement general recursion.

Our implementation of NbE for λ_{sec} extends NbE for Moggi's computational metalanguage [Filinski 2001; Lindley 2005] with a family of monads parameterized by a pre-ordered set of labels. This resembles the parameterization of monads by effects specified by a pre-ordered monoid, also known as *graded monads* [Orchard and Petricek 2014; Wadler 1998], and thus indicates the extensibility of our NbE algorithm to calculi with graded monads. It would be interesting to see if our proof technique can be used to prove noninterference for static enforcement of IFC using graded monads.

Using static semantics means that our work lays a foundation for static analysis of noninterference-like security properties. This opens up a plethora of exciting opportunities for future work. For example, one possibility would be to use type-direction partial evaluation [Danvy 1998] to simplify programs and inspect the resulting programs to verify if they violate security properties. Another arena would be the extension of our proof to more expressive IFC calculi such as DCC or MAC [Vassena et al. 2018]. The main challenge here would be to identify the appropriate static semantics of the language, as they may not always have been designed with one in mind.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G Riecke. 1999. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 147–160.

Andreas Abel and Christian Sattler. 2019. Normalization by Evaluation for Call-by-Push-Value and Polarized Lambda-Calculus. *arXiv preprint arXiv:1902.06097* (2019).

Maximilian Algehed and Jean-Philippe Bernardy. 2019. Simple Noninterference from Parametricity. (2019).

Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical reconstruction of a reduction free normalization proof. In *International Conference on Category Theory and Computer Science*. Springer, 182–199.

Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, Vol. 4. 49.

Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalization by evaluation. In *Prospects for Hardware Foundations*. Springer, 117–137.

William J Bowman and Amal Ahmed. 2015. Noninterference for free. *ACM SIGPLAN Notices* 50, 9 (2015), 101–113.

Catarina Coquand. 1993. From semantics to rules: A machine assisted analysis. In *International Workshop on Computer Science Logic*. Springer, 91–105.

Olivier Danvy. 1998. Type-directed partial evaluation. In *DIKU International Summer School*. Springer, 367–411.

Olivier Danvy, Morten Rhiger, and Kristoffer H Rose. 2001. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming* 11, 6 (2001), 673–680.

Dorothy E Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (1976), 236–243.

Andrzej Filinski. 2001. Normalization by evaluation for the computational lambda-calculus. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 151–165.

GA Kavvos. 2019. Modalities, cohesion, and information flow. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 20.

Peng Li and Steve Zdancewic. 2010. Arrows for secure information flow. *Theoretical computer science* 411, 19 (2010), 1974–1994.

Sam Lindley. 2005. Normalisation by evaluation in the compilation of typed functional programming languages. (2005).

Conor McBride. 2018. Everybody's got to be somewhere. *Electronic Proceedings in Theoretical Computer Science* 275 (2018), 53–69.

Kenji Miyamoto and Atsushi Igarashi. 2004. A modal foundation for secure information flow. In *Workshop on Foundations of Computer Security*. 187–203.

Eugenio Moggi. 1989. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. IEEE, 14–23.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

Dominic A Orchard and Tomas Petricek. 2014. Embedding effect systems in Haskell. (2014).

Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

Gordon Plotkin. 1973. *Lambda-definability and logical relations*. Edinburgh University.

Alejandro Russo, Koen Claessen, and John Hughes. 2009. A library for light-weight information-flow security in Haskell. *ACM Sigplan Notices* 44, 2 (2009), 13–24.

Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *ACM Sigplan Notices*, Vol. 46. ACM, 95–106.

David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe haskell. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 137–148.

Anne Sjerp Troelstra and Helmut Schwichtenberg. 2000. *Basic proof theory*. Number 43. Cambridge University Press.

Stephen Tse and Steve Zdancewic. 2004. Translating dependency into parametricity. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 115–125.

Nachiappan Valliappan and Alejandro Russo. 2019. Exponential Elimination for Bicartesian Closed Categorical Combinators. (2019).

Marco Vassena and Alejandro Russo. 2016. On formalizing information-flow control libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 15–28.

Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2018. Mac a verified static information-flow control library. *Journal of logical and algebraic methods in programming* 95 (2018), 148–180.

Philip Wadler. 1998. The marriage of effects and monads. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 63–74.

A Appendix

A.1 NbE for sums

It is tempting to interpret sums component-wise like products and functions as: $[\![\ \tau_1 + \tau_2\ ]\!] = [\![\ \tau_1\ ]\!] \uplus [\![\ \tau_2\ ]\!]$. However, this interpretation makes it impossible to implement reflection faithfully: should the reflection of a variable $x : \tau_1 + \tau_2$ be a semantic value of type $[\![\ \tau_1\ ]\!]$ (left injection) or $[\![\ \tau_2\ ]\!]$ (right injection)? We cannot make this decision since the value which substitutes x may be either of these cases. The standard solution to this issue is to interpret sums using *decision trees* [Abel and Sattler 2019]. A decision tree allows us to defer this decision until more information is available about the injection of the actual value.

As in the previous case for the monadic type T, a decision tree can be defined as an inductive data type D parameterized by some type interpretation a with the following constructors:

$$\textsc{Leaf}\ \frac{x : a}{\mathsf{leaf}\ x : D\ a} \qquad \textsc{Branch}\ \frac{n : \mathsf{Ne}\ (\tau_1 + \tau_2) \quad f : \mathsf{Var}\ \tau_1 \ \rightarrow\ D\ a \quad g : \mathsf{Var}\ \tau_2\ \rightarrow\ D\ a}{\mathsf{branch}\ n\ f\ g : D\ a}$$

The leaf constructor constructs a leaf of the tree from a semantic value, while the branch constructor constructs a tree which represents a suspended decision over the value of a sum type. The branch constructor is the semantic equivalent of case in normal forms.

Decision trees allow us to model semantic sum values, and hence allow the interpretation of the sum type as follows:

$$[\![\ \tau_1 + \tau_2\ ]\!] \ = \ D\,([\![\ \tau_1\ ]\!] \uplus [\![\ \tau_2\ ]\!])$$

We interpret a sum type (in λ_{sec}) as a decision tree which contains a value of the sum type (in Agda).

As an example, the term *false* of type Bool, implemented as left (), will be interpreted as a decision tree leaf (inj_1 tt) of type D $[\![$ Bool $]\!]$ since we know the exact injection. The Agda constructor inj_1 denotes the left injection in Agda, and inj_2 the right injection. For a variable x of type Bool, however, we cannot interpret it as a leaf since we don't know the actual injection that may substitute it. Instead, it is interpreted as a decision tree by branching over the possible values as branch x (λ _ $\rightarrow$ leaf (inj_1 tt)) (λ _ $\rightarrow$ leaf (inj_2 tt))[7]—which intuitively represents the following tree:

x : Bool

false *true*

In light of this interpretation of sums, the implementation of evaluation for injections is straight-forward since we only need to wrap the appropriate injection inside a leaf:

$$\begin{aligned} &\mathsf{eval}\ (\mathsf{left}\ t)\ \ \ \gamma \ = \ \mathsf{leaf}\ (\mathsf{inj}_1\ (\mathsf{eval}\ t\ \gamma)) \\ &\mathsf{eval}\ (\mathsf{right}\ t)\ \gamma \ = \ \mathsf{leaf}\ (\mathsf{inj}_2\ (\mathsf{eval}\ t\ \gamma)) \end{aligned}$$

For evaluating case however, we must first implement a decision procedure since case is used to make a choice over sums.

To make a decision over a tree of type D $[\![\ \tau\ ]\!]$, we need a function mkDec : D $[\![\ \tau\ ]\!] \ \rightarrow\ [\![\ \tau\ ]\!]$. It can be implemented by induction on the type τ using monadic functions fmap and join on trees, which can in turn be implemented by straight-forward structural induction on the tree. Additionally, we will also need a function which converts a decision over normal forms to a normal form: convert : D (Nf τ) $\rightarrow$ Nf τ. The implementation of this function is made possible by the fact that branch resembles case in normal forms, and can hence be translated to it. We skip the implementation of these functions here, but encourage the reader to see the Agda implementation.

Using these definitions, we can now complete evaluation as follows:

$$\begin{aligned} &\mathsf{eval}\ (\mathsf{case}\ t\ (\mathsf{left}\ x_1 \rightarrow t_1)\ (\mathsf{right}\ x_2 \rightarrow t_2))\ \gamma \ = \\ &\quad \mathsf{mkDec}\ (\mathsf{fmap}\ \mathsf{match}\ (\mathsf{eval}\ t\ \gamma)) \\ &\quad \mathsf{where} \\ &\qquad \mathsf{match} : ([\![\ \tau_1\ ]\!] \uplus [\![\ \tau_2\ ]\!]) \ \rightarrow\ [\![\ \tau\ ]\!] \\ &\qquad \mathsf{match}\ (\mathsf{inj}_1\ v) \ = \ \mathsf{eval}\ t_1\ (\gamma\ [x_1 \mapsto v]) \\ &\qquad \mathsf{match}\ (\mathsf{inj}_2\ v) \ = \ \mathsf{eval}\ t_2\ (\gamma\ [x_2 \mapsto v]) \end{aligned}$$

We first evaluate the term t of type $\tau_1 + \tau_2$ to obtain a tree of type D $([\![\ \tau_1\ ]\!] \uplus [\![\ \tau_2\ ]\!])$. Then, we map the function match which eliminates the sum inside the decision tree to $[\![\ \tau\ ]\!]$, to

[7] We ignore the argument (as λ _) here since it has the uninteresting type ()

produce a tree of type $D \llbracket \tau \rrbracket$. Finally, we run the decision procedure mkDec on the resulting decision tree to produce the desired value of type $\llbracket \tau \rrbracket$.

Reflection for a neutral of a sum type can now be implemented using branch as follows:

```
reflect {τ1 + τ2} n =
  branch n
    (leaf (λ x1 → inj1 (reflect {τ1} x1)))
    (leaf (λ x2 → inj2 (reflect {τ2} x2)))
```

As discussed earlier, we construct the decision tree for neutral n using branch. The subtrees represent all possible semantic values of n and are constructed by reflecting the variables x_1 and x_2.

The function reifyVal, on the other hand, is implemented similar to evaluation by eliminating the sum value inside the decision tree into normal forms as follows:

```
reifyVal {τ1 + τ2} tr = convert (fmap matchNf tr)
  where
    matchNf : (⟦ τ1 ⟧ + ⟦ τ2 ⟧) → Nf (τ1 + τ2)
    matchNf (inj1 x) = left (reifyVal {τ1} x)
    matchNf (inj2 y) = right (reifyVal {τ2} y)
```

With this function, we have completed the implementation of NbE for sums.