Simple Noninterference by Normalization

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

Nachiappan Valliappan
Chalmers University of Technology
nacval@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 • Security and privacy → Formal security models; • Theory of computation → Type theory;

Keywords information-flow control, noninterference, normalization by evaluation

ACM Reference Format:

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 $\text{Int}_H$ denotes a secret integer and $\text{Bool}_I$ denotes a public boolean. Now consider a program $f$ with the following type:

$$f : \text{Int}_H \rightarrow \text{Bool}_I$$

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 [Tse and Zdancewic 2004; Bowman and Ahmed 2015; Algehed and Bernardy 2019]. 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 $\lambda x. \text{true}$ or $\lambda x. \text{false}$; 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 $\lambda x. (\text{not} \ x) \text{true}$ or perhaps the more complex function $\lambda x. (\lambda y. \text{snd} \ (x \ y)) \text{true}$. Observe, however, that both these programs can be normalized to the equivalent function $\lambda x. \text{true}$. In general, although terms in the language may be arbitrarily complex, their normal forms (such as $\lambda x. \text{true}$) 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 $\lambda\text{sec}$, based on Haskell’s seclib library by Russo et al. We present the typing rules and static semantics for $\lambda\text{sec}$ by extending Moggi’s computational metalanguage [Moggi 1991] (Section 2). We
identify normal forms of $\lambda_{sec}$, 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 $\lambda \, x. \, \text{true}$ or $\lambda \, x. \, \text{false}$—as expected (Section 4).

To prove noninterference for all terms using normal forms, we implement normalization for $\lambda_{sec}$ using normalization by 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 $\lambda_{sec}$ (Section 6) —including, for example, a program which operates on both secret and public values such as $\text{Bool}_s \times \text{Bool}_H \rightarrow \text{Bool}_s \times \text{Bool}_H$. 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 $\lambda_{sec}$. Since the proof is primarily type and syntax-directed, it provides an appealing alternative to typical semantics based proof methods. All the main theorems in this paper have been mechanized in the proof assistant Agda\(^1\).

2 The $\lambda_{sec}$ calculus

In this section we present $\lambda_{sec}$, a static IFC calculus that we shall use as the basis for our proof of noninterference. It models the pure and terminating 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:

```haskell
data S (ℓ :: Lattice) a
  return :: a → S ℓ a
  (⇒=) :: S ℓ a → (a → S ℓ b) → S ℓ b
  up :: ℓ_L ⊑ ℓ_H → S ℓ_L a → S ℓ_H a
```

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]) $\langle L, H \rangle$ where the only disallowed flow is from secret ($H$) to public ($L$), denoted $H \not\subseteq L$. The following program written using the seclib API is well-typed and—intuitively—secure:

```haskell
example :: S L Bool → S H Bool
example p = up (p ⇒= λ b → return b)
```

The function example negates the $\text{Bool}$ 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.

**The calculus.** $\lambda_{sec}$ is a simply typed $\lambda$-calculus (STLC) with a 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 $\text{Label}$). 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 $\ell_L$ and $\ell_H$ and refer to them as $\text{public}$ and $\text{secret}$, although they represent levels in an arbitrary security lattice such that $\ell_H \not\subseteq \ell_L$. Figure 1 defines the syntax of terms, types and contexts of $\lambda_{sec}$.

---

\[\begin{array}{l}
\text{Label} \ell, \ell_L, \ell_H , \ell_L \\
\text{Context} \Gamma \Delta \Sigma ::= \emptyset | \Gamma , x : \tau \\
\text{Type} \tau \, t_1 \, t_2 ::= \tau_1 \Rightarrow \tau_2 | \text{true} | () \\
& \ | \tau_1 + \tau_2 | \tau_1 \times \tau_2 \\
& \ | S \, \ell \, \tau \\
\text{Term} \, t \, s \, u ::= x | \lambda \, \ell \, x : \tau | t \, s | () \\
& \ | < t , s > | \text{fst} \, t | \text{snd} \, t \\
& \ | \text{left} \, t | \text{right} \, t \\
& \ | \text{case} \ t \, (\text{left} \, x_1 \rightarrow s) \, (\text{right} \, x_2 \rightarrow u) \\
& \ | \text{return} \ t | \text{let} \, x = t \, \text{in} \, u | \text{up} \ t
\end{array}\]

---

**Figure 1.** The $\lambda_{sec}$ calculus.

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

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

Further, the calculus models the $\text{up}$ combinator in seclib as the construct $\text{up}$. Its purpose is to relabel computations to higher security levels. The rule $[\text{Up}]$, shown in Figure 2, statically enforces that information can only flow from $\ell_L$ to $\ell_H$ in agreement with the security policy $\ell_L \not\subseteq \ell_H$. The rest of the typing rules for $\lambda_{sec}$ are standard [Pierce 2002], and

---

\(^1\)https://github.com/carlostome/ni-nbe

\(^2\)https://hackage.haskell.org/package/seclib
thus omitted here. For a full account we refer the reader to our Agda formalization.

For completeness, the function \( \text{example} \) from earlier can be encoded in the \( \lambda_{\text{sec}} \) calculus as follows:\(^3\)

\[
\text{example} = \lambda \, s. \, \text{up} \left( \text{let } b = s \text{ in } \text{return} \left( \text{not } b \right) \right)
\]

**Static semantics.** The static semantics of \( \lambda_{\text{sec}} \) is defined as a set of equations relating terms of the same type typed under the same environment. The equations characterize pairs of \( \lambda_{\text{sec}} \) terms that are equivalent based on \( \beta \)-reduction, \( \eta \)-expansion and other monadic operations. We present the equations for \text{return} and \text{let} constructs of the monadic type \( S \) (à la Moggi [1991]) in Figure 3, and further extend this with equations for the \text{up} primitive in Figure 4. The remaining equations—including \( \beta \) and \( \eta \) rules for other types, and permutation rules for commuting case conversions—are fairly standard [Lindley 2005; Abel and Sattler 2019], 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 \( t_1 \).

\[
\Gamma \vdash t_1 \approx t_2 : \tau
\]

\( \beta \)-S

\[
\Gamma \vdash t_1 : \tau \quad \Gamma, x : \tau \vdash t_2 : S \ell \tau
\]

\( \eta \)-S

\[
\Gamma, x : \tau \approx t \text{ in } t_2 \approx t_1 [x/t_1] : S \ell \tau
\]

\( \gamma \)-S

\[
\Gamma \vdash t : S \ell \tau
\]

\[
\Gamma \vdash t \approx \text{let } x = t \text{ in } \left( \text{return } x \right) : S \ell \tau
\]

\[
\Gamma \vdash t_1 : S \ell \tau_1 \quad \Gamma, x : \tau_1 \vdash t_2 : S \ell \tau_2
\]

\[
\Gamma, x : \tau_1, y : \tau_2 \vdash t_3 : S \ell \tau_3
\]

\[
\Gamma \vdash \text{let } x = \left( \text{let } y = t_1 \text{ in } t_2 \right) \text{ in } t_3 \approx \text{let } y = \left( \text{let } x = t_2 \text{ in } t_1 \right) : S \ell \tau_3
\]

**Figure 3.** Static semantics of \( \lambda_{\text{sec}} \) (return and let).

\( \delta_1 \)-S

\[
\Gamma \vdash t : S \ell_1 \tau_1
\]

\[
\Gamma, x : \tau_1 \uparrow u : S \ell_2 \tau_2
\]

\[
\Gamma \vdash \text{up}_p \left( \text{let } x = t \text{ in } u \right) \approx \text{let } x = \left( \text{up}_p \right) t \text{ in } \left( \text{up}_p \right) u : S \ell_H \tau
\]

**Figure 4.** Static semantics of \( \lambda_{\text{sec}} \) (up).

\( \delta_2 \)-S

\[
\Gamma \vdash t : \tau
\]

\[
p : \ell_L \subseteq \ell_H
\]

\[
\Gamma \vdash \text{up}_p \left( \text{return}_L t \right) \approx \text{return}_H t : S \ell_H \tau
\]

\( \delta_{\text{TRANS}} \)-S

\[
\Gamma \vdash t : S \ell_1 \tau
\]

\[
p : \ell_L \subseteq \ell_M
\]

\[
q : \ell_M \subseteq \ell_H
\]

\[
r = \text{FlowsToTrans } p q
\]

\[
\Gamma \vdash \text{up}_q \left( \text{up}_p t \right) \approx \text{up}_r \left( \text{up}_p t \right) : S \ell_H \tau
\]

\( \delta_{\text{REFL}} \)-S

\[
\Gamma \vdash t : \ell_1 \tau
\]

\[
p : \ell_H \subseteq \ell
\]

\[
\Gamma \vdash \text{up}_p \left( \text{up}_p t \right) \approx \text{up}_r \left( \text{up}_p t \right) : S \ell_H \tau
\]

3 In \( \lambda_{\text{sec}} \), the type \( \text{Bool} \) is encoded as \( () + () \) with \( \text{false} = \text{left } () \) and \( \text{true} = \text{right } () \).
A subformula of a type is defined as:

$$\text{Subformula}$$

can also be defined for types as follows:

For example, given a neutral form of type $$\Gamma \vdash \tau$$, neutrality states that the type $$\tau$$ must occur as a subformula of some type in the typing context $$\Gamma$$. An example of such a context is $$\Gamma = [x : () \Rightarrow \text{Bool} , y : S \ell_H t]$$. 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 $$\tau, \tau_1$$ and $$\tau_2$$; a subformula of a type is defined as:

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

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

**Normal forms.** Intuitively, normal forms of type $$\Gamma \vdash \tau$$ are characterized as terms of type $$\Gamma \vdash \tau$$ that cannot be reduced further using the static semantics. Precisely, a normal form is a term obtained by systematically applying the equations defined by the relation $$\approx$$ in a specific order to a given term. We leave the exact order of applying the equations unspecified here since we only require that there exists a normal form for every term—we prove this later in Section 5. The normal forms in Figure 6 extend the $$\beta$$-short $$\eta$$-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 $$\text{left()}$$ (which denotes the value $$\text{false}$$) of type $$\emptyset \vdash \text{Bool}$$ constructs a value of the type $$\text{Bool}$$ in the empty context $$\emptyset$$.

The reader may have noticed that the $$\text{let}$$ 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 $$\text{let}$$ can be quoted by first applying $$\text{up}$$ to the quotation of the neutral and then using $$\text{let}$$. The reason $$\text{let}$$ represents both $$\text{let}$$ and $$\text{up}$$ in the normal forms is to retain the non-reducibility of normal forms. Had we added $$\text{up}$$ separately to normal forms, then this may trigger further reductions. For example, the term $$\text{up}(\text{return}())$$ can be reduced further to the term $$\text{return}()$$. Disallowing $$\text{up}$$-terms directly in normal forms disallows the possibility of this reduction in normal forms. Similarly, adding $$\text{up}$$ 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 $$\emptyset \vdash \tau$$ for any type $$\tau$$. By neutrality, if such a neutral form exists, then $$\tau$$ must be a subformula of the empty context $$\emptyset$$, but this is impossible! Similarly, the $$\eta$$-long form of normal forms guarantees that a normal form of a function type must begin with either a $$\lambda$$ or $$\text{case}$$—hence reducing the number of possible cases in our proof. In the next section, we utilize these properties to show that the program $$f$$ (from earlier) behaves as a constant.
4 Normal Forms and Noninterference

The program $f : \text{Int} \rightarrow \text{Bool}$ from Section 1 can be generalized in $\lambda_{\text{sec}}$ as a term$^4$ $\emptyset + f : S \ell_H \tau \Rightarrow S \ell_L \text{Bool}$ 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 $\lambda_{\text{sec}}$—states that given two levels $\ell_L$ (public) and $\ell_H$ (secret) such that the flow of information from secret to public is disallowed as $\ell_H \not\subseteq \ell_L$; 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.

As explained earlier, for $\emptyset + f : S \ell_H \tau \Rightarrow S \ell_L \text{Bool}$ to satisfy noninterference, it must be equivalent to the constant function whose body is return true or return false independent 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 $\emptyset \rnf f : S \ell_H \tau \Rightarrow S \ell_L \text{Bool}$, either $f \equiv \lambda x. \text{return true}$ or $f \equiv \lambda x. \text{return false}$.

Note that the equality relation $\equiv$ denotes syntactic (or propositional) equality, which means that the normal forms on both sides must be syntactically identical. The proof follows by direct case analysis on the normal forms of type $\emptyset \rnf f : S \ell_H \tau \Rightarrow S \ell_L \text{Bool}$:

Proof of Lemma 4.1. Upon closer inspection of the normal forms of $\lambda_{\text{sec}}$ (Figure 6), the reader may notice that for the function type $\emptyset \rnf f : S \ell_H \tau \Rightarrow S \ell_L \text{Bool}$ there exists only two possibilities: a case or a $\lambda$ construct. The former, can be easily dismissed by neutrality because the scrutinee—a neutral form of sum type $t_1 + t_2$—to appear in the empty context. In the latter case, the $\lambda$ 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 $\emptyset , x : S \ell_H \tau \rnf _ : S \ell_L \text{Bool}$.

Considering the normal forms of type $\emptyset , x : S \ell_H \tau \rnf _ : S \ell_L \text{Bool}$, 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 $\emptyset , x : S \ell_H \tau$. Analogously, the monadic let with a neutral term of type $S \ell_L \tau$, expects this type to occur in the same context—but it does not, since $S \ell_L \tau$ is not a subformula of $S \ell_H \tau$. The remaining case, return, can be further refined, where the only possibilities leave us with $\lambda \ x . \ (\text{return true})$ or $\lambda \ x . \ (\text{return false})$. 

In order to show that noninterference holds for arbitrary programs of type $\emptyset + f : S \ell_H \tau \Rightarrow S \ell_L \text{Bool}$ 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 $\lambda_{\text{sec}}$ 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 $\text{eval}$, and the extraction of normal forms, called reification, is implemented by an inverse function $\text{reify}$. Normalization is implemented as a function from terms to normal forms by composing these functions:

\[
\begin{align*}
\text{norm} : (\Gamma \vdash \tau) & \rightarrow (\Gamma \rnf \tau) \\
\text{norm} \ t & = \text{reify} (\text{eval} t)
\end{align*}
\]

The function $\text{eval}$ and $\text{reify}$ have the following types in the host language:

\[
\begin{align*}
\text{eval} : (\Gamma \vdash \tau) & \rightarrow (\Gamma \Sigma) \\
\text{reify} : (\Gamma \Sigma) & \rightarrow (\Gamma \rnf \tau)
\end{align*}
\]

In these types, the function $\Gamma \Sigma$ interprets types and contexts in $\lambda_{\text{sec}}$ as types in the host language. That is, the type $\Gamma \Sigma$ denotes the interpretation of the $(\lambda_{\text{sec}})$ type $\tau$ in the host language, and similarly for $\Gamma \Sigma$. On the other hand, the function $\Gamma \Sigma \rightarrow \Gamma \Sigma$—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 $\lambda_{\text{sec}}$. Similarly, we also get a proof that normalization terminates for free since Agda ensures that all functions are terminating.

We implement the functions $\text{eval}$ and $\text{reify}$ for terms in $\lambda_{\text{sec}}$, 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].

$^4\lambda_{\text{sec}}$ does not have polymorphic types, in this case $\tau$ represents an arbitrary but concrete type, for instance unit (1).
In the remainder of this section, we will denote the typing derivations \( \Gamma \vdash_{\text{nf}} \tau \) and \( \Gamma \vdash_{\text{ne}} \tau \) as \( \text{NF} \tau \) and \( \text{Ne} \tau \) respectively. We leave the context \( \Gamma \) 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 \( \text{Var} \tau \), leaving \( \Gamma \) 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 \( i, () \times \) and \( \Rightarrow \). The implementation for sums is a bit more technical, and hence deferred to Appendix A.1. 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:

- \([ i ] = \text{NF} i\)
- \([ () ] = \tau\)
- \([ \tau_1 \times \tau_2 ] = [ \tau_1 ] \times [ \tau_2 ]\)
- \([ \tau_1 \Rightarrow \tau_2 ] = \{ [ \tau_1 ] \to [ \tau_2 ] \} \)

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

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

- \([ \emptyset ] = \emptyset\)
- \([ \Gamma, x : \tau_1 ] = [ \Gamma ] [ \text{Var} \tau_1 \mapsto [ \tau_1 ] ]\)

For example, a value \( \gamma \) which inhabits the interpretation \([ \Gamma ]\) denotes the execution environment for evaluating a term typed in the context \( \Gamma \).

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

- \( \text{eval} \ x \quad \gamma = \text{lookup} \ x \ \gamma \)
- \( \text{eval} \ () \quad \gamma = \text{tt} \)
- \( \text{eval} \ (\text{fst} \ t) \quad \gamma = \pi_1 (\text{eval} \ t \ \gamma) \)
- \( \text{eval} \ (\text{snd} \ t) \quad \gamma = \pi_2 (\text{eval} \ t \ \gamma) \)
- \( \text{eval} \ (\langle \tau_1, \tau_2 \rangle) \quad \gamma = (\text{eval} \ t_1 \ \gamma, \text{eval} \ t_2 \ \gamma) \)
- \( \text{eval} \ (\lambda \ x. t) \quad \gamma = \lambda \ v \rightarrow \text{eval} \ t \ (\gamma [x \mapsto v]) \)
- \( \text{eval} \ (t \ s) \quad \gamma = (\text{eval} \ t \ \gamma) (\text{eval} \ s \ \gamma) \)

Note that \( \gamma \) is an execution environment for the term’s context; \( \text{lookup} \) and \( \pi_1 \) and \( \pi_2 \) are Agda functions; and \( \text{tt} \) is the constructor of the unit type \( \tau \). 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 \( \gamma \) 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 \( \text{reflect} \) and \( \text{reifyVal} \). The function \( \text{reflect} \) converts neutral forms to semantic values, while the dual function \( \text{reifyVal} \) converts semantic values to normal forms. These functions are implemented as follows:

\[
\begin{align*}
\text{reifyVal} : [\tau] &\to \text{NF} \tau \\
\text{reifyVal} \ {\langle i \rangle \ n} &\ = n \\
\text{reifyVal} \ {\langle () \rangle \ tt} &\ = () \\
\text{reifyVal} \ {\langle \tau_1 \times \tau_2 \rangle \ p} &\ = \\
\text{< \text{reflect} \ {\tau_1} \ (\text{fst} \ n), \text{reflect} \ {\tau_2} \ (\text{snd} \ n)>} \\
\text{reifyVal} \ {\langle \tau_1 \Rightarrow \tau_2 \rangle \ n} &\ = \\
\lambda \ x. \text{reifyVal} \ {\tau_2} \ (f \ (\text{reflect} \ {\tau_1} \ x)) \ | \ \text{fresh} \ x \\
\end{align*}
\]

\[
\begin{align*}
\text{reflect} : \text{Ne} \ \tau &\to [\tau] \\
\text{reflect} \ {\langle i \rangle \ n} &\ = n \\
\text{reflect} \ {\langle () \rangle \ tt} &\ = \text{tt} \\
\text{reflect} \ {\langle \tau_1 \times \tau_2 \rangle \ n} &\ = \\
\text{reflect} \ {\langle \tau_1 \Rightarrow \tau_2 \rangle \ n} &\ = \\
\lambda \ v \rightarrow \text{reflect} \ {\tau_2} \ (n \ (\text{reifyVal} \ {\tau_1} \ v)) \\
\end{align*}
\]

Note that the argument inside the braces \( \{ \} \) denotes an implicit parameter, which is the type of the corresponding neutral/value argument of \( \text{reflect/reifyVal} \) 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 \( \text{reifyVal} \) is also implemented in a similar fashion by induction on types.

To implement reification, recollect that the argument to \( \text{reify} \) is a function that results from partially applying the \( \text{eval} \) 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:

\[
\begin{align*}
\text{genEnv} : (\Gamma : \text{Ctx}) &\to [\Gamma] \\
\text{genEnv} \ \emptyset &\ = \emptyset \\
\text{genEnv} \ (\Gamma, x : \tau) &\ = \text{genEnv} \ \Gamma \ [x \mapsto \text{reflect} \ x] \\
\end{align*}
\]

Finally, we can now implement \( \text{reify} \) as follows:

\[
\begin{align*}
\text{reify} \ {\langle \Gamma \rangle \ f} &\ = \text{let} \ \gamma = \text{genEnv} \ \Gamma \ \text{in} \ \text{reifyVal} \ (f \ \gamma) \\
\end{align*}
\]
We generate an environment $\gamma$ to apply the semantic function $f$, and then convert the resulting semantic value to a normal form by applying $\text{reifyVal}$.

### 5.2 NbE for the security monad

To interpret a type $S \ell \tau$, 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 $\ell$ and some type $a$ (which would be $\llbracket \tau \rrbracket$ in this case). Evidently this monad must allow the implementation of the semantic counterparts of the terms $\text{return}$, $\text{let}$ and $\text{up}$ in $\lambda_{\text{sec}}$ as follows:

$$
\begin{align*}
\text{return} & : a \rightarrow T \ell a \\
\text{bind} & : T \ell a \rightarrow (a \rightarrow T \ell b) \rightarrow T \ell b \\
\text{up} & : (\ell_L \sqsubseteq \ell_H) \rightarrow T \ell_L a \rightarrow T \ell_H a
\end{align*}
$$

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

$$
\text{Return} \quad \begin{array}{c}
x : a \\
\text{return } x : T \ell a
\end{array}
$$

<table>
<thead>
<tr>
<th>BINDN</th>
<th>$p : \ell_L \sqsubseteq \ell_H$</th>
<th>$n : \text{Ne } \ell_L \tau$</th>
<th>$f : \text{Var } \tau \rightarrow T \ell_H a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{bindNe } p \ n \ f : T \ell_H a$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

$$
\begin{align*}
\text{up } p \ (\text{return } v) & = \\
& \text{return } v \\
\text{up } p \ (\text{bindNe } q \ n \ f) & = \\
& \text{bindNe } (\text{trans } q \ p) \ n \ (\lambda x \rightarrow \text{up } p \ (f \ x))
\end{align*}
$$

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

$$
\llbracket S \ell \tau \rrbracket = T \ell \llbracket \tau \rrbracket
$$

Having mirrored the monadic primitives in $\lambda_{\text{sec}}$ using semantic counterparts, evaluation is rather simple:

$$
\begin{align*}
\text{eval } (\text{return } t) \gamma & = \text{return } (\text{eval } t \gamma) \\
\text{eval } (\text{up } p \ t) \gamma & = \text{up } p \ (\text{eval } t \gamma) \\
\text{eval } (\text{let } x = t \text{ in } s) \gamma & = \\
& \text{bind } (\text{eval } t \gamma) \ (\lambda v \rightarrow \text{eval } s \ (\gamma \ [x \mapsto v]))
\end{align*}
$$

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

$$
\begin{align*}
\text{reflect } \{ S \ell \tau \} \ n = \\
& \text{bindNe } (\lambda x \rightarrow \text{reflect } \{ \tau \} \ x)
\end{align*}
$$

Since we do not need to increase the sensitivity of the neutral to bind it here, we simply provide the "reflexive flow"

$$
\begin{align*}
\text{reifyVal } \{ S \ell \tau \} \ (\text{return } v) & = \\
& \text{return } (\text{reifyVal } \{ \tau \} \ v) \\
\text{reifyVal } \{ S \ell \tau \} \ (\text{bindNe } \{ p \} \ n \ f) & = \\
& \text{let } \{ p \} x = n \text{ in } \text{reifyVal } \{ \tau \} \ (f \ x)
\end{align*}
$$

### 5.3 Preservation of semantics

To prove that normalization preserves static semantics of $\lambda_{\text{sec}}$, 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 $\Gamma \vdash t \equiv \\text{norm } t \Downarrow : \tau$

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 Agda mechanization of this theorem.

6 Noninterference for \( \lambda_{\text{sec}} \)

After developing the necessary machinery to normalize terms in the calculus, we are ready to state and prove noninterference for \( \lambda_{\text{sec}} \). 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 \( \ell_1 \) and \( \ell_2 \) such that \( \ell_1 \not\subseteq \ell_2 \) and a function \( \emptyset \vdash f : S_{\ell_1} \tau \Rightarrow S_{\ell_2} \text{Bool} \) then \( \forall s_1 s_2 : S_{\ell_1} \tau, f s_1 \approx f s_2 \).

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 \( \emptyset \vdash f : S_{\ell_1} \tau \Rightarrow S_{\ell_2} \text{Bool} \) 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 \( \text{norm } f \). In this manner, we obtain that the normal forms of \( f \) are exactly the constant function that returns \textit{true} or \textit{false} wrapped in the \textit{return}. In the former case, by correctness of normalization we have that \( f \approx \text{true} \) \text{norm } f \approx \lambda x. \text{return true}. \) By \( \beta \)-reduction and congruence of term-level function application, we have that \( \forall t. (\lambda x. \text{return true}) t \approx \text{return true}. \) Therefore, \( f s_1 \approx f s_2 \). The case when \( \text{norm } f \equiv \lambda x. \text{return false} \) follows a similar argument. \( \square \)

The noninterference property proven above characterizes what it means for a concrete class of programs, i.e. those of type \( \emptyset \vdash f : S_{\ell_1} \tau \Rightarrow S_{\ell_2} \text{Bool} \), to be secure: the attacker cannot learn one bit of the secret from using program \( f \). Albeit interesting, this property does not scale to more complex programs; for instance if the function \( f \) was typed in a non empty context the proof of the above lemma would not hold. The rest of this section is dedicated to generalize and prove noninterference from the program \( f \) to arbitrary programs written in \( \lambda_{\text{sec}} \). As will become clear, normal forms of \( \lambda_{\text{sec}} \) play a crucial role towards proving noninterference.

6.2 General Noninterference theorem

In order to discuss general noninterference for \( \lambda_{\text{sec}} \), we must first specify what are the secret \( (\ell_{\text{IH}}) \) inputs of a program and its public \( (\ell_L) \) output with respect to an attacker at level \( \ell_L \). 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., \textit{transparent} and \textit{ground}, to the attacker.

**Definition 6.1** (Transparent type).

- (\( \emptyset \)) is transparent at any level \( \ell \).
- (\( \iota \)) is transparent at any level \( \ell \).
- (\( \tau_1 \Rightarrow \tau_2 \)) is transparent at \( \ell \) iff \( \tau_2 \) is transparent at \( \ell \).
- (\( \tau_1 + \tau_2 \)) is transparent at \( \ell \) iff \( \tau_1 \) and \( \tau_2 \) are transparent at \( \ell \).
- (\( \tau_1 \times \tau_2 \)) is transparent at \( \ell \) iff \( \tau_1 \) and \( \tau_2 \) are transparent at \( \ell \).
- (\( S \ell' \tau \)) is transparent at \( \ell \) iff \( \ell' \subseteq \ell \) and \( \tau \) is transparent at \( \ell \).

**Definition 6.2** (Ground type).

- (\( \emptyset \)) is ground.
- (\( \iota \)) is ground.
- (\( \tau_1 + \tau_2 \)) is ground iff \( \tau_1 \) and \( \tau_2 \) are ground.
- (\( \tau_1 \times \tau_2 \)) is ground iff \( \tau_1 \) and \( \tau_2 \) are ground.
- (\( S \ell \tau \)) is ground iff \( \tau \) is ground.

A type \( \tau \) is transparent at security level \( \ell \) if the type does not include the security monad type over a higher security level \( \ell_{\text{IH}} \). A ground type, on the other hand, is a first order type, i.e. a type that does not contain a function type. These simplifying restrictions over the output type of a program allow us to state a generic noninterference property over terms and perform induction on the normal forms.

These restrictions do not hinder the generality of our security condition: a program producing a partially public output, for instance a product \( S \ell_L \text{Bool} \times S \ell_{\text{IH}} \text{Bool} \), can be transformed to produce a fully public output by applying the \textit{snd} projection. We return to this example later at the end of the section. Also note that previous work on proving noninterference for static IFC languages [Abadi et al. 1999; Miyamoto and Igarashi 2004] also impose similar restrictions.

Departing from the traditional view of programs as closed terms, i.e. terms without free variables, in the \( \lambda_{\text{sec}} \) calculus we consider all terms for which a typing derivation exists. 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 be secret at level \( \ell \). These definitions, dubbed \( \ell \)-sensitivity, force the types appearing in the context to be at least as sensitive as \( \ell \).

**Definition 6.3** (Context sensitivity).

A context \( \Gamma \) is \( \ell \)-sensitive if and only if for all types \( \tau \in \Gamma \), \( \tau \) is \( \ell \)-sensitive. A type \( \tau \) is \( \ell \)-sensitive, on the other hand, if and only if:

- \( \tau \) is the function type \( \tau_1 \Rightarrow \tau_2 \) and \( \tau_2 \) is \( \ell \)-sensitive.
- \( \tau \) is the product type \( \tau_1 \times \tau_2 \) and \( \tau_1 \) and \( \tau_2 \) are \( \ell \)-sensitive.
- \( \tau \) is the monadic type \( S \ell' \tau_1 \) and \( \ell \not\subseteq \ell' \).
Next, we define substitutions\(^5\), which lay at the core of \(\beta\)-reduction rules in the \(\lambda_{sec}\) calculus. Substitutions map free variables in a term to other terms possibly typed in a different context.

\[
\text{Substitution } \sigma := \sigma_0 \mid \sigma \ [x \mapsto t]
\]

\[
\begin{array}{ll}
\Gamma \vdash_{\sub} \sigma : \Delta & \Gamma \vdash_{\sub} \Gamma \vdash t : \tau \\
\Gamma \vdash_{\sub} \sigma [x \mapsto t] : \Delta, x : \tau & \Gamma \vdash_{\sub} \sigma_0 : \emptyset \\
\end{array}
\]

Figure 7. Substitutions of \(\lambda_{sec}\)

A substitution is either empty, \(\sigma_0\), or is the substitution \(\sigma\) extended with a new mapping from the variable \(x : \tau\) to term \(t\). We denote \([t \ [\sigma] \] the application of substitution \(\sigma\) 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 \(\Gamma\) 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 \(\sigma\) into the composition of a public substitution, \(\Gamma \vdash_{\sub} \sigma_{pl} : \Delta\), that fixes the public inputs, and a secret substitution \(\Delta \vdash_{\sub} \sigma_{se} : \Sigma\), that restricts \(\Delta\) to be \(\ell_H\)-sensitive. The composition of both, denoted \(\Gamma \vdash_{\sub} (\sigma_{pl} ; \sigma_{se}) : \Sigma\), maps variables in context \(\Gamma\) to terms typed in \(\Sigma\); first, \(\sigma_{pl}\) maps variables from \(\Gamma\) to terms in \(\Delta\), subsequently, \(\sigma_{se}\) maps variables in \(\Delta\) to terms typed in \(\Sigma\). Below, we state \(\ell_L\)-equivalence of substitutions:

**Definition 6.4 (Low equivalence of substitutions).**

Two substitutions \(\sigma_1\) and \(\sigma_2\) are \(\ell_L\)-equivalent, written \(\sigma_1 \equiv_{\ell_L} \sigma_2\), if and only if for all \(\ell_H\) such that \(\ell_H \notin \ell_L\), there exists a public substitution \(\sigma_{pl}\), and two secret substitutions \(\sigma_{se}^1\) and \(\sigma_{se}^2\) such that \(\sigma_1 \equiv_{\ell_H} \sigma_{pl} ; \sigma_{se}^1\) and \(\sigma_2 \equiv_{\ell_H} \sigma_{pl} ; \sigma_{se}^2\).

Informally, noninterference for \(\lambda_{sec}\) 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 of a type at level \(\ell_L\), typed in a \(\ell_H\)-sensitive context, must be constant. We achieve this in turn by showing that the normal forms of such terms are constant. Below, we state what it means for a term to be constant:

\(^5\)In Section 2 we purposely left capture-avoiding substitutions underspecified, we amend that here.

**Definition 6.5 (Constant term).**

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

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 \(x : i \Rightarrow i\cdot y : i\cdot \tau\) by the identity function is not a normal form—and cannot be derived syntactically as a normal form using \(\tau_{\sub}\). Instead, we lean on the shape of the context to state the property.

If a normal form \(\Gamma \vdash_{\sub} n : \tau\) is constant, then there must exist a syntactically identical derivation \(\emptyset \vdash_{\sub} n' : \tau\) such that \(n \equiv n'\). However, since \(n\) and \(n'\) are typed in different contexts, \(\Gamma\) and \(\emptyset\), it is not possible to compare them for syntactic equality. We solve this problem by renaming the normal form \(n'\) to add as many variables as mentioned in context \(\Gamma\). The signature of the renaming function is the following:

\[
\text{ren} : (\Gamma \leq \Delta) \rightarrow (\Gamma \vdash_{\sub} \tau) \rightarrow (\Delta \vdash_{\sub} \tau)
\]

The relation \(\leq\) between contexts \(\Gamma\) and \(\Delta\) indicates that the variables appearing in \(\Delta\) are at least those present in \(\Gamma\). This relation, called weakening, is defined as follows:

- \(\emptyset \leq \emptyset\)
- If \(\Gamma \leq \Delta\), then \(\Gamma \leq \Delta, x : \tau\)
- If \(\Gamma \leq \Delta\), then \(\Gamma, x : \tau \leq \Delta, x : \tau\)

The function \(\text{ren}\) can be defined by simple induction on the derivation of the normal forms. Note that terms can also be renamed in the same fashion.

**Definition 6.6 (Constant normal form).** A normal form \(\Gamma \vdash_{\sub} n : \tau\) is constant if there exists a normal form \(\emptyset \vdash_{\sub} n' : \tau\) such that \(\text{ren}(n') \equiv n\).

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 \(\emptyset \vdash_{\sub} n' : \tau\) such that \(\text{ren}(n') \equiv n\). Let the quotient of this normal form \(\tau\) be some term \(\emptyset \vdash t' : \tau\). Recall from earlier that terms can also be renamed, hence we have \(\text{ren}(t') \approx \text{ren}(\tau n')\) by correctness of \(n'\). Since it can be shown that \(\text{ren}(\tau n') \equiv \tau \approx \text{ren}(\tau n')\), we have that \(\text{ren}(\tau n') \equiv \tau n\), and by correctness of \(n\), we also have \(\text{ren}(t') \approx \tau - (1)\).

A substitution \(\sigma\) maps free variables in a term to terms. The empty substitution, denoted \(\sigma_0\), is the unique substitution, such that \(\Delta \vdash \sigma_0 : \tau\) for any \(\Delta\). That is, applying the empty substitution simply renames the term. We can
show that \( t' [ \sigma_0 ] \equiv \text{ren} (t') \), and hence, by (1), we have \( t' [ \sigma_0 ] \approx t - (2) \). Since \( \sigma_0 \) renames a term typed in the empty context, we can show that for any substitution \( \sigma \), we have \( (t' [ \sigma_0 ]) [ \sigma ] \approx t' [ \sigma_0 ] \). Because \( \sigma_0 \) is also unique, for any two substitutions \( \sigma_1 \) and \( \sigma_2 \), we have \( (t' [ \sigma_0 ]) [ \sigma_1 ] \approx (t' [ \sigma_0 ]) [ \sigma_2 ] \) 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. \( \square \)

The key insight of our noninterference proof is reflected in the following lemma which shows how normal forms of \( \lambda_{\text{sec}} \) typed in a sensitive context are either constant or the flow between the security level of the context and the output type is permitted. Below we include the proof to showcase how it follows by straightforward induction on the shape of the normal forms.

**Lemma 6.3** (Normal forms do not leak). Given a normal form \( \Gamma \vdash_{\text{nf}} n : \tau \), where the context \( \Gamma \) is \( \ell_1 \)-sensitive, and \( \tau \) is a ground and transparent type at level \( \ell_o \), then either \( n \) is constant or \( \ell_i \subseteq \ell_o \).

**Proof.** By induction on the structure of the normal form \( n \). Note that \( \lambda \) and \( \text{case} \) normal forms need not be considered since the preconditions ensure that \( \tau \) cannot be a function type (dismisses \( \lambda \)), and \( \Gamma \) cannot contain a variable of a sum type (dismisses \( \text{case} \)).

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

The last step to noninterference is an ancillary lemma which shows that terms typed in \( \ell_1 \)-sensitive contexts are constant:

**Lemma 6.4**. Given a term \( \Gamma \vdash t : \tau \), where the context \( \Gamma \) is \( \ell_1 \)-sensitive, and \( \tau \) is a ground type transparent at \( \ell_1 \). If \( \ell_1 \not\subseteq \ell_o \), then \( t \) is constant.

The proof follows from lemmas Lemma 6.3 and Lemma 6.2.

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

**Theorem 6.5** (Noninterference for \( \lambda_{\text{sec}} \)). Given security levels \( \ell_1 \) and \( \ell_2 \) such that \( \ell_1 \not\subseteq \ell_2 \), an attacker at level \( \ell_1 \); two \( \ell_1 \)-equivalent substitutions \( \sigma_1 \) and \( \sigma_2 \) such that \( \sigma_1 \approx_{\ell_1} \sigma_2 \); and a type \( \tau \) that is ground and transparent at \( \ell_1 \); 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 \( \sigma_1 \approx_{\ell_1} \sigma_2 \) gives that \( \sigma_1 = \sigma_{\ell_1} : \sigma_{\ell_1}^1 \) and \( \sigma_2 = \sigma_{\ell_1} : \sigma_{\ell_1}^2 \). After applying the public substitution \( \sigma_{\ell_1} \) to the term \( \Gamma \vdash t : \tau \), we are left with a term typed in a \( \ell_2 \)-sensitive context \( \Delta \vdash t [ \sigma_{\ell_1} ] : \tau \). By Lemma 6.4, \( t [ \sigma_{\ell_1} ] \) is constant which means that \( t [ \sigma_{\ell_1} ] [ \sigma_{\ell_1}^1 ] \approx (t [ \sigma_{\ell_1} ]) [ \sigma_{\ell_1}^2 ] \). By readjusting substitutions using composition we obtain \( t [ \sigma_{\ell_1} ] [ \sigma_{\ell_1}^1 ] \approx (t [ \sigma_{\ell_1} ]) [ \sigma_{\ell_1}^2 ] \), 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 \( \lambda_{\text{sec}} \) for programs of type \( 0 + t : S \ell_1 \text{Bool} \times S \ell_2 \text{Bool} \Rightarrow \Gamma \ell_1 \text{Bool} \times S \ell_2 \text{Bool} \), which are the recurring example for explaining noninterference in the literature [Russo et al. 2009; Bowman and Ahmed 2015]. 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 \( \ell_1 \) and \( \ell_2 \) such that \( \ell_1 \not\subseteq \ell_2 \) and a program \( x : S \ell_1 \text{Bool} \times S \ell_2 \text{Bool} \vdash t : S \ell_1 \text{Bool} \times S \ell_2 \text{Bool} \) then \( \forall p : S \ell_1 \text{Bool} \times s_1 \times s_2 : S \ell_2 \text{Bool} \), 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 \text{snd} projection. This is justified because the first component of the output is protected at level \( \ell_1 \), which the attacker cannot observe. Below we prove noninterference for \( x : S \ell_1 \text{Bool} \times S \ell_2 \text{Bool} \vdash \text{snd} \vdash t : S \ell_2 \text{Bool} \).

**Proof of Corollary 6.6.** To apply Theorem 6.5 we have to show that both substitutions are equivalent, \( x \mapsto (p, s) \)
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 [Wadler 1998; Orchard and Petricek 2014], 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

We thank Alejando Russo, Fabian Ruch, Sandro Stucki and Maximilian Algehed for the insightful discussions on normalization and noninterference. We would also like to thank Irene Lobo Valbuena, Claudio Agustin Mista and the anonymous reviewers at PLAS’19 for their comments on earlier drafts of this paper. This work was funded by the Swedish Foundation for Strategic Research (SSF) under the projects WebSec (Ref. RIT17-0011) and Octopi (Ref. RIT17-0023).

References


A Appendix

A.1 NbE for sums

It is tempting to interpret sums component-wise like products and functions as: \[
\left\{ \tau_1 + \tau_2 \right\} = \left\{ \tau_1 \right\} \cup \left\{ \tau_2 \right\}.
\]
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 \( \left\{ \tau_1 \right\} \) (left injection) or \( \left\{ \tau_2 \right\} \) (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:

<table>
<thead>
<tr>
<th>LEAF</th>
<th>BRANCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x : \tau )</td>
<td>( n : \text{Ne} (\tau_1 + \tau_2) )</td>
</tr>
<tr>
<td>( f : \text{Var} \tau_1 \rightarrow D )</td>
<td>( g : \text{Var} \tau_2 \rightarrow D )</td>
</tr>
</tbody>
</table>

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

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

\[
\left\{ \tau_1 + \tau_2 \right\} = D \left( \left\{ \tau_1 \right\} \cup \left\{ \tau_2 \right\} \right)
\]

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

As an example, the term \( \text{false} \) of type \( \text{Bool} \), implemented as \( \text{left} () \), will be interpreted as a decision tree leaf \( \text{inj}_1 \) of type \( D \left[ \text{Bool} \right] \) since we know the exact injection. The Agda constructor \( \text{inj}_1 \) denotes the left injection in Agda, and \( \text{inj}_2 \) the right injection. For a variable \( x \) of type \( \text{Bool} \), 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 \( \text{branch} x \left( \lambda y \rightarrow \text{leaf} \left( \text{inj}_1 \, y \right) \right) \left( \lambda y \rightarrow \text{leaf} \left( \text{inj}_2 \, y \right) \right) \) — which intuitively represents the following tree:

\[
\begin{array}{c}
\text{false} \\
\text{true}
\end{array}
\]

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

\[
\begin{align*}
\text{eval} \left( \text{left} \, t \right) \, y &= \text{leaf} \left( \text{inj}_1 \, \left( \text{eval} \, t \, y \right) \right) \\
\text{eval} \left( \text{right} \, t \right) \, y &= \text{leaf} \left( \text{inj}_2 \, \left( \text{eval} \, t \, y \right) \right)
\end{align*}
\]

\(^6\)We ignore the argument \( (\lambda) \) here since it has the uninteresting type \( () \)
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 $D \llbracket r \rrbracket$, we need a function $\text{mkDec} : D \llbracket r \rrbracket \to \llbracket r \rrbracket$. It can be implemented by induction on the type $r$ using monadic functions $\text{fmap}$ and $\text{join}$ on trees, which can in turn be implemented by straightforward structural induction on the tree. Additionally, we will also need a function which converts a decision over normal forms to a normal form: $\text{convert} : D (\text{NF } r) \to \text{NF } r$.

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:

\[
\text{eval (case } t \text{ (left } x_1 \to t_1 \text{) (right } x_2 \to t_2) } \gamma = \\
\text{mkDec (fmap } \text{match (eval } t \gamma \text{)})
\]

where

\[
\begin{align*}
\text{match} & : (\llbracket r_1 \rrbracket \sqcup \llbracket r_2 \rrbracket) \to \llbracket r \rrbracket \\
\text{match} (\text{inj}_1 v) & = \text{eval } t_1 (\gamma [x_1 \mapsto v]) \\
\text{match} (\text{inj}_2 v) & = \text{eval } t_2 (\gamma [x_2 \mapsto v])
\end{align*}
\]

We first evaluate the term $t$ of type $r_1 + r_2$ to obtain a tree of type $D (\llbracket r_1 \rrbracket \sqcup \llbracket r_2 \rrbracket)$. Then, we map the function match which eliminates the sum inside the decision tree to $\llbracket r \rrbracket$, to produce a tree of type $D \llbracket r \rrbracket$. Finally, we run the decision procedure $\text{mkDec}$ on the resulting decision tree to produce the desired value of type $\llbracket r \rrbracket$.

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

\[
\text{reflect } \{ r_1 + r_2 \} n = \\
\text{branch } n \\
(\text{leaf } (\lambda x_1 \to \text{inj}_1 (\text{reflect } \{ r_1 \} x_1))) \\
(\text{leaf } (\lambda x_2 \to \text{inj}_2 (\text{reflect } \{ r_2 \} x_2)))
\]

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 $\text{reifyVal}$, on the other hand, is implemented similar to evaluation by eliminating the sum value inside the decision tree into normal forms as follows:

\[
\text{reifyVal } \{ r_1 + r_2 \} tr = \text{convert (fmap } \text{matchNf } tr) \\
\text{where}
\begin{align*}
\text{matchNf} & : (\llbracket r_1 \rrbracket \sqcup \llbracket r_2 \rrbracket) \to \text{NF } (r_1 + r_2) \\
\text{matchNf} (\text{inj}_1 x) & = \text{left } (\text{reifyVal } \{ r_1 \} x) \\
\text{matchNf} (\text{inj}_2 y) & = \text{right } (\text{reifyVal } \{ r_2 \} y)
\end{align*}
\]

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