A Modal Foundation for Secure Information Flow

Kenji Miyamoto Atsushi Igarashi
Graduate School of Informatics, Kyoto University
{miyamoto, igarashi}@kuis.kyoto-u.ac.jp

Abstract

Information flow analysis is a program analysis that detects possible illegal information flow such as the leakage of (partial information on) passwords to the public. Recently, several type-based techniques for information flow analysis have been proposed for various kinds of programming languages. Details of those type systems, however, vary substantially and even their core parts are often slightly different, making the essence of the type system unclear.

In this paper we propose a typed lambda calculus $\lambda_s^{\Box}$ as a foundation for information flow analysis. The type system is developed so that it corresponds to a proof system of an intuitionistic modal logic of validity by the Curry-Howard isomorphism. The calculus enjoys the properties of subject reduction, confluence, and strong normalization. Moreover, we give a very simple proof of the noninterference property, which guarantees that, in a well-typed program, no information on confidential data is leaked to the public. We also demonstrate that a core part of the SLam calculus by Heintze and Riecke can be encoded into $\lambda_s^{\Box}$.

Keywords: Curry-Howard isomorphism, information flow analysis, modal logic, noninterference, and type systems.

1 Introduction

Background. Increasing demands for security in software have recently been stimulating the research on language-based security. Among such work is a program analysis technique called *information flow analysis* [8, 24], which statically checks whether or not secret information, such as passwords, is leaked by program execution.

Information flow analysis keeps track of how information on confidential data flows. In the literature, information flow is classified into two: *explicit* and *implicit* flow. Explicit flow occurs when, for example, confidential data are assigned to public variables while implicit flow arises from the control structure of a program: it occurs when confidential data control which conditional branch to take. For example, consider the following program fragment:

```
pub := if length(passwd) > 5 then 1 else 2;
```

and suppose `pub` is a public variable, `passwd` is a confidential string, and `length` is a function to compute the length of a given string. In information flow analysis, this program is considered insecure—even though `passwd` itself is not leaked, its partial information, that is, whether `passwd` is more than five characters, is leaked to `pub`. A principal correctness property of information flow analysis is called *noninterference*. This property intuitively means that, given a program that takes a confidential input and yields a public output, the output remains the same no matter what value is given as the input.

Recently, a lot of *type-based* techniques for information flow analysis have been proposed for various kinds of languages, including procedural [27, 26], functional [10, 1, 23], object-oriented [19, 3, 4] and, concurrent

languages [25, 11, 12, 22, 15]. The basic idea of type-based analysis is as follows: (1) types are extended so that they include security information as well as standard information on the kinds of values, such as *int* or *int* $\rightarrow$ *int*; (2) typing rules are constructed by taking security information into account—for example, a conditional expression whose test involves a high-security type is well typed only if both branches are given high-security types to prevent implicit flow; and, finally, (3) a type reconstruction algorithm for the type system is developed. Type-based information flow analysis has been drawing much attention, partially because it cleanly separates the specification and algorithm of the analysis as a type system and type reconstruction, respectively.

Details of those type systems, however, vary substantially (apart from the difference of their base languages) and even their functional core parts are often slightly different. Also, fairly complicated techniques are used to prove noninterference (especially in those for functional languages): some use denotational semantics [10, 1, 3] and some use a non-standard operational semantics [23]. It makes it difficult not only to compare those techniques but also to grasp the essence of the type system and the noninterference property.

Our Goal and Approach. Our goal here is to give a foundational account for type-based information flow analysis. To achieve this goal, we, inspired by the following observation, develop a natural extension of the Curry-Howard isomorphism between a type system for information flow analysis and a certain modal logic.

Modal logic is a language to talk about truth relative to time or places, etc., which are abstracted as possible worlds. Here, we talk about things like at what level the information on certain values can be available. So, it seems to natural to relate the notion of security levels to possible worlds. Since information available at a lower level is also available at a higher level, a suitable modality seems to be *local validity* (or *validity* for simplicity) $\Box_\ell$, which means "it is true at every level higher (or equal to) the security level ℓ that ..." So, the fact that ℓ_1 is higher than ℓ_2 (or information can flow from ℓ_2 to ℓ_1) can be regarded as that a possible world ℓ_1 is reachable from ℓ_2. It is expected that the proposition $\Box_\ell A$ naturally corresponds to the type that represents the values of type A at the security level ℓ.

Our Contributions. The contributions we make in this paper are summarized as follows:

- As discussed above, we point out an informal correspondence between a type system for information flow analysis and a certain modal logic.
- To make this correspondence more formal, we develop a typed λ-calculus $\lambda_s^\Box$ with *modal types* of the form $\Box_\ell A$ and prove that it enjoys subject reduction, Church-Rosser, and strong normalization.
- We also prove noninterference, which is essential to information flow analysis. Our proof is very simple, without using complex denotational techniques or non-standard reduction relations as in previous work.
- To demonstrate that $\lambda_s^\Box$ can be a foundation for information flow analysis, we develop encoding from (a purely-functional subset of) the SLam calculus [10] to $\lambda_s^\Box$.

Structure of the paper The rest of the paper is organized as follows. Section 2 introduces $\lambda_s^\Box$ with its syntax, type system and reduction. Section 3 proves its properties including noninterference. After showing how the SLam calculus can be encoded to $\lambda_s^\Box$ in Section 4, we discuss related work in Section 5 and give concluding remarks in Section 6. Most of the proofs are omitted for brevity.

2 The System $\lambda_s^\Box$

In this section, we define the typed λ-calculus $\lambda_s^\Box$. We first briefly introduce a proof system of the modal logic discussed in the previous section and then proceed to the formal definition of $\lambda_s^\Box$.

2.1 Modal Logic of Local Validity

The proof system is partially inspired by Pfenning and Davies' formalization [21], based on the notion of judgments.

The basic idea is to consider judgments to assert truth and local validity separately. Accordingly, a judgment has two kinds of assumption sets and that of truth is written $A_1^{\ell_1}, \ldots, A_n^{\ell_n}; B_1, \ldots, B_m \vdash^{\ell} C$. It means C is true at ℓ under the assumption that A_i is true *everywhere reachable from* ℓ_i and B_j is true at the current level. (In what follows, we write Δ for $A_1^{\ell_1}, \ldots, A_n^{\ell_n}$ and Γ for $B_1, \ldots, B_m$.) A locally valid assumption can be used only when the current level is reachable from the level of the assumption, while the rule for (ordinary) truth assumptions are as usual, as the following two rules:

$$\frac{\ell_i \sqsubseteq \ell}{A_1^{\ell_1}, \ldots, A_i^{\ell_i}, \ldots, A_n^{\ell_n}; \Gamma \vdash^{\ell} A_i} \qquad\qquad \Delta; B_1, \ldots, B_j, \ldots, B_m \vdash^{\ell} B_j$$

Validity is expressed by a judgment of truth with zero truth assumptions. The introduction rule for $\Box_\ell A$ amounts to internalizing a judgment of validity as a proposition and the elimination rule turns "$\Box_\ell A$ is true" into "A is valid at ℓ":

$$\frac{\Delta; \cdot \vdash^{\ell} C}{\Delta; \Gamma \vdash^{\ell'} \Box_\ell C} \qquad\qquad \frac{\Delta; \Gamma \vdash^{\ell'} \Box_\ell A \qquad \Delta, A^{\ell}; \Gamma \vdash^{\ell'} C}{\Delta; \Gamma \vdash^{\ell'} C}$$

The rules for other logical connectives are straightforward. For example, the elimination rule of implication is as follows.

$$\frac{\Delta; \Gamma \vdash^{\ell} A \to B \qquad \Delta; \Gamma \vdash^{\ell} A}{\Delta; \Gamma \vdash^{\ell} B}$$

Note that the levels of the judgments must be the same.

Keeping this in mind, we proceed to the formal definition of our calculus.

2.2 Syntax

We first assume the countably infinite set **OVar** of *ordinary variables*, ranged over by x, y, z, and **MVar** of *modal variables*, ranged over by u and v. We also assume the partially ordered set $(\mathcal{L}, \sqsubseteq)$ of levels; elements of $\mathcal{L}$ are ranged over by ℓ.

The types of $\lambda_s^{\Box}$ are simple types with the unit type, product types, sum types, and modal types.

2.2.1 Definition [Types]: The set of types, ranged over by A, B, and C, is defined by the following grammar:

$$A ::= \mathit{unit} \mid A \to A \mid A \times A \mid A + A \mid \Box_\ell A$$

The precedence of type constructor is given by the decreasing order $\Box_\ell > \times > + > \to$ and the function type constructor is right associative. For example, $A \to \Box_\ell B \to A \times C$ stands for $A \to ((\Box_\ell B) \to (A \times C))$ and $\Box_\ell A \times \Box_\ell C + B \to A$ for $(((\Box_\ell A) \times (\Box_\ell C)) + B) \to A$.

2.2.2 Definition [Terms]: The set of terms, ranged over by M and N, is defined by the grammar:

$$\begin{array}{rcl} M & ::= & x \mid u \mid () \mid \lambda x : A.M \mid M\, M \mid \langle M, M \rangle \mid \pi_1(M) \mid \pi_2(M) \mid \iota_1(M) \mid \iota_2(M) \\ & \mid & (\textbf{case}\ M\ \textbf{of}\ \iota_1(x) \Rightarrow M \mid \iota_2(x) \Rightarrow M) \mid \textbf{box}_\ell\, M \mid \textbf{let box}_\ell\, u = M\ \textbf{in}\ M \end{array}$$

We omit parentheses according to the usual convention and assume that application is left associative. Also, bound variables and their scopes are defined in a customary manner: $\lambda x : A.M$ bounds x in M; **case** M **of** $\iota_1(x_1) \Rightarrow N_1 \mid \iota_2(x_2) \Rightarrow N_2$ bounds x_1 and x_2 in N_1 and N_2, respectively; **let box**$_\ell\, u = M$ **in** N bounds u in N.

Terms are mostly those of the simply typed λ-calculus with unit, products, and sums. We have two kinds of term variables as the logic has two kinds of assumptions: truth and validity. As we shall see in typing rules, **box**$_\ell\, M$ corresponds to an application of the introduction rule for $\Box_\ell$ when viewed as a proof term, and can be thought as a *sealing* operation where the sealed value is accessed at security level ℓ. Similarly, **let box**$_\ell\, u = M$ **in** N corresponds to an application of the elimination rule, when viewed as a proof term, and can be thought as unsealing. Operationally, if M reduces to **box**$_\ell\, M_0$, then **let box**$_\ell\, u = M$ **in** N reduces to N in which M_0 is substituted for u.

Free variables are defined as usual except that there are two kinds of variables. We write *FMV*(M) (*FOV*(M), resp.) for modal (ordinary, resp.) variables that occur free in M. For example, *FMV*$(\langle \iota_2(z), \lambda x : A.x\, u\rangle) = \{u\}$ and *FOV*$(\langle \iota_2(z), \lambda x : A.x\, u\rangle) = \{z\}$ and *FMV*(**let box**$_\ell\, u = x\, v$ **in** $u\, y$) $= \{v\}$.

We write $[M/x]$ ($[M/u]$, resp.) for the capture-avoiding substitution of M for the ordinary variable x (the modal variable u, resp.).

2.3 Type System

As discussed above, the judgment form of the logic is $A_1^{\ell_1}, \ldots, A_n^{\ell_n}; B_1, \ldots, B_m \vdash^\ell C$. Accordingly, the type judgment of $\lambda_s^\Box$ is of the form $\Delta; \Gamma \vdash^\ell M : A$, read as "$M$ is given type A at level ℓ under modal context Δ and ordinary context Γ." A modal context, which corresponds to $A_1^{\ell_1}, \ldots, A_n^{\ell_n}$, is a sequence of the form $u ::^\ell A$ where modal variables in the sequence are pairwise distinct. Similarly, an ordinary context, which corresponds to $B_1, \ldots, B_m$, is a sequence of the form $x : B$ where variables in the sequence are pairwise distinct. We often write '$\cdot$' for the empty modal/ordinary context. When both contexts in a judgment are empty, they are simply omitted and the judgment is written $\vdash^\ell M : A$. We write $u ::^\ell A \in \Delta$ when Δ includes $u ::^\ell A$. Similarly for ordinary contexts.

The whole set of typing rules is given in Figure 1. We explain key rules in detail; other rules are fairly standard (modulo two kinds contexts and the annotation of a level).

The rule T-OVAR for ordinary variable reference is just as usual. On the other hand, modal variables can be used only when the level of the variable is lower than or equal to the current level (the rule T-MVAR). From the viewpoint of information flow, a program at a lower level cannot refer to a variable of a higher security level, preventing confidential information from flowing into irrelevant levels. It may first appear that the rule T-MVAR is too strict because a term containing a (modal) variable of high security as a free variable is regarded as confidential computation even if the variable is just discarded. However, with a certain programming style, we can remedy it: (a core of) the SLam calculus can be encoded into $\lambda_s^\Box$ as we discuss in Section 4. In this sense, $\lambda_s^\Box$ is as expressive as the SLam calculus.

The rule T-BOX introduces modal types.Since "$\Box_\ell A$ is true" stands for "A is true everywhere reachable from ℓ," the premise must be a judgment of validity, that is, the ordinary context must be empty. Since the judgment does not depend on any assumptions of a particular level, it holds at any level, hence ℓ'. For weakening, any ordinary context can be placed in the conclusion. With the terminology of information flow, a sealed piece of code may be used at an arbitrary level above or equal to ℓ, so it cannot refer to (ordinary) variables available only at a particular security level.

The last rule T-LETBOX eliminates modal types. It turns "$\Box_\ell A$ is true" into "A is valid at ℓ" and adds $u ::^\ell A$ to the modal context to deduce B. It might look odd that ℓ is not necessarily related to ℓ' at which M is unsealed. Actual access restriction is enforced by T-MVAR and, if u is used in N (not under **box**$_\ell$), it should be the case that $\ell \sqsubseteq \ell'$.

$$\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash^{\ell} x : A} \quad \text{(T-OVAR)} \qquad \frac{\Delta; \Gamma \vdash^{\ell} M : A_1 \times A_2 \qquad i \in \{1,2\}}{\Delta; \Gamma \vdash^{\ell} \pi_i(M) : A_i} \quad \text{(T-PROJ)}$$

$$\frac{u ::^{\ell'} A \in \Delta \qquad \ell' \sqsubseteq \ell}{\Delta; \Gamma \vdash^{\ell} u : A} \quad \text{(T-MVAR)} \qquad \frac{\Delta; \Gamma \vdash^{\ell} M : A_i \qquad i \in \{1,2\}}{\Delta; \Gamma \vdash^{\ell} \iota_i(M) : A_1 + A_2} \quad \text{(T-INJ)}$$

$$\Delta; \Gamma \vdash^{\ell} () : \mathit{unit} \quad \text{(T-UNIT)} \qquad \frac{\Delta; \Gamma \vdash^{\ell} M : A_1 + A_2 \quad \Delta; \Gamma, x_1 : A_1 \vdash^{\ell} N_1 : B \quad \Delta; \Gamma, x_2 : A_2 \vdash^{\ell} N_2 : B}{\Delta; \Gamma \vdash^{\ell} \mathbf{case}\ M\ \mathbf{of}\ \iota_1(x_1) \Rightarrow N_1 \mid \iota_2(x_2) \Rightarrow N_2 : B} \quad \text{(T-CASE)}$$

$$\frac{\Delta; \Gamma, x : A \vdash^{\ell} M : B}{\Delta; \Gamma \vdash^{\ell} \lambda x : A.M : A \to B} \quad \text{(T-ABS)}$$

$$\frac{\Delta; \Gamma \vdash^{\ell} M : A \to B \qquad \Delta; \Gamma \vdash^{\ell} N : A}{\Delta; \Gamma \vdash^{\ell} M\, N : B} \quad \text{(T-APP)} \qquad \frac{\Delta; \cdot \vdash^{\ell'} M : A}{\Delta; \Gamma \vdash^{\ell} \mathbf{box}_{\ell'}\, M : \Box_{\ell'} A} \quad \text{(T-BOX)}$$

$$\frac{\Delta; \Gamma \vdash^{\ell} M : A \qquad \Delta; \Gamma \vdash^{\ell} N : B}{\Delta; \Gamma \vdash^{\ell} \langle M, N \rangle : A \times B} \quad \text{(T-PAIR)} \qquad \frac{\Delta; \Gamma \vdash^{\ell} M : \Box_{\ell'} A \qquad \Delta, u ::^{\ell'} A; \Gamma \vdash^{\ell} N : B}{\Delta; \Gamma \vdash^{\ell} \mathbf{let\ box}_{\ell'}\, u = M\ \mathbf{in}\ N : B} \quad \text{(T-LETBOX)}$$

Figure 1: Typing Rules of $\lambda_s^{\Box}$

2.3.1 Example: If $\ell \sqsubseteq \ell'$, then the type judgment $\vdash^{\ell'} \lambda x : \Box_{\ell} A.\mathbf{let\ box}_{\ell}\, u = x\ \mathbf{in}\ u : \Box_{\ell} A \to A$ can be derived. By ignoring levels, this type can be viewed as to a variant of the axiom M.

2.3.2 Example: If $\ell_1 \sqsubseteq \ell_3$ and $\ell_2 \sqsubseteq \ell_3$, then

$$\vdash^{\ell} \lambda x : \Box_{\ell_1}(A \to B).\mathbf{let\ box}_{\ell_2}\, u = x\ \mathbf{in}\ \lambda y : \Box_{\ell_2} A.\mathbf{let\ box}_{\ell_2}\, v = y\ \mathbf{in\ box}_{\ell_3}\,(u\, v) \\ : \Box_{\ell_1}(A \to B) \to \Box_{\ell_2} A \to \Box_{\ell_3} B$$

is derivable. This type corresponds to a variant of the axiom K.

2.4 Operational Semantics

Operational semantics is given by the reduction relation of the form $M \longrightarrow M'$, read as "M reduces to M' in one step." The computation rules are given as follows.

$$\begin{array}{rcl} (\lambda x : A.M_1)\, M_2 & \longrightarrow & [M_2/x]M_1 \\ \pi_i(\langle M_1, M_2 \rangle) & \longrightarrow & M_i \\ \mathbf{case}\ \iota_i(M)\ \mathbf{of}\ \iota_1(x_1) \Rightarrow N_1 \mid \iota_2(x_2) \Rightarrow N_2 & \longrightarrow & [M/x_i]N_i \\ \mathbf{let\ box}_{\ell}\, u = \mathbf{box}_{\ell}\, M\ \mathbf{in}\ N & \longrightarrow & [M/u]N \end{array}$$

They can be applied at any point in a term, so we also need the obvious congruence rules (if $M_1 \longrightarrow M_1'$, then $M_1\, M_2 \longrightarrow M_1'\, M_2$, and the like), which we omit here.

2.4.1 Example: Let $M = (\lambda x : \Box_{\ell} A.\mathbf{let\ box}_{\ell}\, u = x\ \mathbf{in}\ \pi_2(u))\,(\mathbf{box}_{\ell}\,\langle M_1, M_2 \rangle)$. Then, M reduces to M_2 as follows:

$$\begin{array}{rcl} M & \longrightarrow & [\mathbf{box}_{\ell}\,\langle M_1, M_2 \rangle / x]\mathbf{let\ box}_{\ell}\, u = x\ \mathbf{in}\ \pi_2(u) \quad (= \mathbf{let\ box}_{\ell}\, u = \mathbf{box}_{\ell}\,\langle M_1, M_2 \rangle\ \mathbf{in}\ \pi_2(u)) \\ & \longrightarrow & [\langle M_1, M_2 \rangle / u]\pi_2(u) \quad (= \pi_2(\langle M_1, M_2 \rangle)) \longrightarrow M_2 \end{array}$$

3 Properties of $\lambda_s^{\Box}$

In this section, we show that $\lambda_s^{\Box}$ satisfies basic properties including subject reduction, Church-Rosser and strong normalization. Then, we show that it also satisfies the noninterference property, as expected.

3.1 Basic Properties

All the statements of the properties mentioned above are standard. Note that, in Subject Reduction, not only is the type of a term preserved during reduction but also is the level at which the type judgments are derived. They are proved by standard techniques.

3.1.1 Theorem [Subject Reduction]: If $\Delta; \Gamma \vdash^{\ell} M : A$ and $M \longrightarrow N$ then $\Delta; \Gamma \vdash^{\ell} N : A$.

3.1.2 Theorem [Church-Rosser]: If $M \xrightarrow{*} M_1$ and $M \xrightarrow{*} M_2$, then there exists a term N such that $M_1 \xrightarrow{*} N$ and $M_2 \xrightarrow{*} N$.

3.1.3 Theorem [Strong normalization]: If $\Delta; \Gamma \vdash M : A$, then M is strongly normalizing.

3.2 Noninterference

One of the most important correctness properties is *noninterference*, which intuitively means that a program input at a high security level does not affect the program output at a lower level. To state this property more formally, we require the following technical definition.

3.2.1 Definition [Transparent ground types]: A type A is *transparent ground type at level* ℓ if and only if:

1. $A =$ *unit*,
2. $A = A_1 \times A_2$ and both A_1 and A_2 are transparent ground types at ℓ,
3. $A = A_1 + A_2$ and both A_1 and A_2 are transparent ground types at ℓ, or
4. $A = \Box_{\ell'} A_0$ and $\ell' \sqsubseteq \ell$ and A_0 is transparent ground type at ℓ'.

Intuitively, a transparent ground type at ℓ represents values of which it is effectively possible to inspect equality.

Now, the noninterference property is stated as follows:

3.2.2 Theorem [Noninterference]: If $u ::^{\ell} A; \cdot \vdash^{\ell'} M : B$ and B is a transparent ground type at ℓ' and $\ell \not\sqsubseteq \ell'$, then, there exists a unique normal form M' (modulo α-conversion) such that, for any N, if $\vdash^{\ell} N : A$, then $[N/u]M \xrightarrow{*} M'$.

In this statement, u serves as a high level input, whose information is not allowed to flow into the level ℓ' (hence $\ell \not\sqsubseteq \ell'$). The condition on B expresses the fact that the value of B can be inspected (or observed) at level ℓ'. Thus, it cannot include function types or modal types at a level irrelevant to ℓ'.

This theorem can be proved by a very simple manner: it turns out that the modal variable that stands for a high level input will disappear during reduction—this is shown simply by inspecting the structure of normal forms (Theorem 3.2.4). Then, Theorem 3.2.2 is obtained as an easy corollary. We sketch the proof below.

First, we introduce neutral terms which correspond to applications of the elimination rules.

3.2.3 Definition [Neutral terms]: A term is *neutral* if it is of the form x, u, $M\,N$, $\pi_i(M)$, **case** M **of** $\iota_1(x_1) \Rightarrow N_1 \mid \iota_2(x_2) \Rightarrow N_2$, or **let box**$_\ell\, u = M$ **in** N.

3.2.4 Theorem: If $u ::^{\ell} A; \cdot \vdash^{\ell'} M : B$ and M is a normal form and B is a transparent ground type at ℓ' and $u \in \mathit{FMV}(M)$, then $\ell \sqsubseteq \ell'$.

Proof: By induction on the derivation of $u ::^{\ell} A; \cdot \vdash^{\ell'} M : B$ with case analysis on the last rule used. □

Finally, we can prove Theorem 3.2.2.

Proof of Theorem 3.2.2: Let M' be a normal form such that $M \xrightarrow{*} M'$. By Theorem 3.1.1, $u ::^{\ell'} A; \cdot \vdash^{\ell} M' : B$. Then, by the assumption $\ell \not\sqsubseteq \ell'$ and (a contraposition of) Theorem 3.2.4, $u \notin \mathit{FMV}(M')$, hence $[N/u]M' = M'$. By using the fact that $M \longrightarrow M'$ implies $[N/u]M \longrightarrow [N/u]M'$, we have $[N/u]M \xrightarrow{*} M'$, which is a normal form. By Theorems 3.1.2 and 3.1.3, any reduction sequence from $[N/u]M$ will end with M'. □

4 Encoding the SLam Calculus

In this section, we show how (a pure fragment of) the SLam calculus [10] can be encoded into $\lambda_s^{\Box}$. Recursive functions have been dropped because the target language $\lambda_s^{\Box}$ is not equipped with them. Also, we have dropped `protect` in the SLam calculus for the following reasons: (1) the static semantics of `protect`, which raises the security level of the type of the operand, can be simulated by application of a coercion function; (2) the dynamic semantics of `protect`, which dynamically raises the security level of a value, is not relevant to ensure noninterference—even if it was "nop," noninterference could be proved. (In fact, coercion functions used in our encoding are essentially identity functions.) We first briefly review the definition of the SLam calculus and then show the translation with its correctness theorems.

4.1 Review of the SLam Calculus

Let $\mathcal{L}$ be a join semilattice of security levels, ranged over by ℓ. The elements of $\mathcal{L}$ are ordered by $\sqsubseteq$ and the binary join of ℓ_1 and ℓ_2 is written $\ell_1 \vee \ell_2$. A type, more precisely a secure type, in the SLam calculus is a simple type where every type constructor is annotated with a security level, which signifies at which level the information on a value of the type may be available.

4.1.1 Definition [SLam types]: The set of *SLam types*, ranged over by t, and the set of *SLam secure types*, ranged over by s are defined as follows:

$$\begin{array}{lcl} t & ::= & \mathit{unit} \mid s \times s \mid s + s \mid s \to s \\ s & ::= & (t, \ell) \end{array}$$

When $s = (t, \ell)$, we often write $s \bullet \ell'$ for $(t, \ell \vee \ell')$ and $\sharp s$ for ℓ.

4.1.2 Definition [SLam expressions]: The set of expressions, ranged over by the metavariable e, are formed by the typing rules in Figure 2.

An operational semantics of the SLam calculus is given by the following computation rules together with congruence rules, omitted for brevity.

$$\begin{array}{lcl} (\lambda x : s.e_0)_{\ell}\, e_1 & \rightsquigarrow & [e_1/x]e_0 \\ \pi_i(\langle e_1, e_2 \rangle_{\ell}) & \rightsquigarrow & e_i \\ \textbf{case}\ \iota_i(e_0)_{\ell}\ \textbf{of}\ \iota_1(x_1) \Rightarrow e_1 \mid \iota_2(x_2) \Rightarrow e_2 & \rightsquigarrow & [e_0/x_i]e_i \end{array}$$

$$\frac{\Gamma(x) = s}{\Gamma \vdash x : s} \qquad \frac{\Gamma \vdash e : s \qquad s \leq s'}{\Gamma \vdash e : s'} \qquad \Gamma \vdash ()_\ell : (\mathit{unit}, \ell) \qquad \frac{\Gamma, x : s_1 \vdash e_0 : s_2}{\Gamma \vdash (\lambda x : s_1.e_0)_\ell : (s_1 \to s_2, \ell)}$$

$$\frac{\Gamma \vdash e_1 : (s_2 \to s_0, \ell) \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash e_1\, e_2 : s_0 \bullet \ell} \qquad \frac{\Gamma \vdash e_1 : s_1 \qquad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2 \rangle_\ell : (s_1 \times s_2, \ell)} \qquad \frac{\Gamma \vdash e : (s_1 \times s_2, \ell)}{\Gamma \vdash \pi_i(e) : s_i \bullet \ell} \qquad \frac{\Gamma \vdash e : s_i}{\Gamma \vdash \iota_i(e)_\ell : (s_1 + s_2, \ell)}$$

$$\frac{\Gamma \vdash e_0 : (s_1 + s_2, \ell) \quad \Gamma, x_1 : s_1 \vdash e_1 : s \qquad \Gamma, x_2 : s_2 \vdash e_2 : s}{\Gamma \vdash \textbf{case}\ e_0\ \textbf{of}\ \iota_1(x_1) \Rightarrow e_1 \mid \iota_2(x_2) \Rightarrow e_2 : s \bullet \ell} \qquad \frac{\ell \sqsubseteq \ell'}{(\mathit{unit}, \ell) \leq (\mathit{unit}, \ell')}$$

$$\frac{\ell \sqsubseteq \ell' \qquad s_1' \leq s_1 \qquad s_2 \leq s_2'}{(s_1 \to s_2, \ell) \leq (s_1' \to s_2', \ell')} \qquad \frac{\ell \sqsubseteq \ell' \qquad s_1 \leq s_1' \qquad s_2 \leq s_2'}{(s_1 \times s_2, \ell) \leq (s_1' \times s_2', \ell')} \qquad \frac{\ell \sqsubseteq \ell' \qquad s_1 \leq s_1' \qquad s_2 \leq s_2'}{(s_1 + s_2, \ell) \leq (s_1' + s_2', \ell')}$$

Figure 2: Typing rules of a core of the SLam calculus

Translation from subtyping to coercion: $s_1 \leq s_2 \searrow M$

$$\frac{\ell \sqsubseteq \ell'}{(\mathit{unit}, \ell) \leq (\mathit{unit}, \ell') \searrow \ \lambda x : \Box_\ell \mathit{unit}.\textbf{let box}_\ell\, u_x = x\ \textbf{in}\ u_x}$$

$$\frac{\ell \sqsubseteq \ell' \qquad s_1' \leq s_1 \searrow M_1 \qquad s_2 \leq s_2' \searrow M_2}{(s_1 \to s_2, \ell) \leq (s_1' \to s_2', \ell') \searrow \ \lambda x : |(s_1 \to s_2, \ell)|.\textbf{let box}_\ell\, u_x = x\ \textbf{in}\ \lambda y : |s_1'|.M_2\, (\textbf{box}_{\sharp s_2}\, (u_x\, (\textbf{box}_{\sharp s_1}\, (M_1\, y))))}$$

$$\frac{\ell \sqsubseteq \ell' \qquad s_1 \leq s_1' \searrow M_1 \qquad s_2 \leq s_2' \searrow M_2}{(s_1 \times s_2, \ell) \leq (s_1' \times s_2', \ell') \searrow \ \lambda x : |(s_1 \times s_2, \ell)|.\textbf{let box}_\ell\, u_x = x\ \textbf{in}\ \langle \textbf{box}_{\sharp s_1'}\, (M_1\, (\pi_1(u_x))), \textbf{box}_{\sharp s_2'}\, (M_2\, (\pi_2(u_x))) \rangle}$$

$$\frac{\ell \sqsubseteq \ell' \qquad s_1 \leq s_1' \searrow M_1 \qquad s_2 \leq s_2' \searrow M_2}{(s_1 + s_2, \ell) \leq (s_1' + s_2', \ell') \searrow \ \lambda x : |(s_1 + s_2, \ell)|.\textbf{let box}_\ell\, u_x = x\ \textbf{in}\ \textbf{case}\ u_x\ \textbf{of}\ \iota_1(y) \Rightarrow \iota_1(\textbf{box}_{\sharp s_1}\, (M_1\, y)) \mid \iota_2(z) \Rightarrow \iota_2(\textbf{box}_{\sharp s_2}\, (M_2\, z))}$$

Figure 3: Translation from SLam to $\lambda_s^\Box$

4.2 Translation from SLam to $\lambda_s^\Box$

The translation from the SLam calculus to $\lambda_s^\Box$ is rather straightforward. We take $\mathcal{L}$ as the partially ordered set of levels. Secure types are translated to ($\lambda_s^\Box$) types by the function $|s|$:

$$|\mathit{unit}| \;=\; \mathit{unit}, \quad |s_1 \mathbf{op} s_2| \;=\; |s_1| \mathbf{op} |s_2|, \quad |(t, \ell)| \;=\; \Box_\ell |t|$$

where **op** is $\times$, $+$, or $\to$. For the sake of simplicity, we assume that all bound variable names are different and there is a bijection from the set of SLam variables to the set of modal variables. The modal variable corresponding to x is written u_x below. Then, type-directed translation rules for expressions are given in Figures 3 and 4.

As will be formally stated in Theorem 4.2.1, a SLam expression of type (t, ℓ) will be translated to a term typed at ℓ. The translation follows the following patterns: (1) when a subexpression is consumed by a given SLam operation (for example, $e_1\, e_2$ consumes e_1 but not e_2), it is translated to the corresponding operation and the result is immediately unsealed; and (2) when a subexpression is not really consumed by an operation (for example, none of function application, pairing, and injection consume their arguments), the subexpression is first sealed and passed to the operation. This straightforward translation can introduce a lot of unsealing followed by sealing; developing an optimized translation may be of interesting.

As mentioned in Section 2, a term containing a (modal) variable of high security as a free variable is regarded as confidential computation even if the variable is just discarded. The translation patterns also show how to avoid

Translation of expressions: $\Gamma \vdash e : s \searrow M$

$$\frac{\Gamma(x) = s}{\Gamma \vdash x : s \searrow u_x} \qquad \frac{\Gamma \vdash e : s \searrow N \qquad s \leq s' \searrow M}{\Gamma \vdash e : s' \searrow M\,(\mathbf{box}_{\sharp s}\, N)} \qquad \Gamma \vdash ()_\ell : (\mathit{unit}, \ell) \searrow ()$$

$$\frac{\Gamma, x : s \vdash e_0 : s_0 \searrow M}{\Gamma \vdash (\lambda x : s.e_0)_\ell : (s \to s_0, \ell) \searrow \lambda x : |s|.\mathbf{let\ box}_\ell\, u_x = x\ \mathbf{in\ box}_{\sharp s_0}\, M}$$

$$\frac{\Gamma \vdash e_1 : (s_2 \to s_0, \ell) \searrow M_1 \qquad \Gamma \vdash e_2 : s_2 \searrow M_2 \qquad u \notin \mathit{FMV}(M_1) \cup \mathit{FMV}(M_2)}{\Gamma \vdash e_1\, e_2 : s_0 \bullet \ell \searrow \mathbf{let\ box}_{\sharp s_0}\, u = M_1\,(\mathbf{box}_{\sharp s_2}\, M_2)\ \mathbf{in}\ u}$$

$$\frac{\Gamma \vdash e_1 : s_1 \searrow M_1 \quad \Gamma \vdash e_2 : s_2 \searrow M_2}{\Gamma \vdash \langle e_1, e_2\rangle_\ell : (s_1 \times s_2, \ell) \searrow \langle \mathbf{box}_{\sharp s_1}\, M_1, \mathbf{box}_{\sharp s_2}\, M_2\rangle} \qquad \frac{\Gamma \vdash e : (s_1 \times s_2, \ell) \searrow M \quad i \in \{1,2\} \quad u \notin \mathit{FMV}(M)}{\Gamma \vdash \pi_i(e) : s_i \bullet \ell \searrow \mathbf{let\ box}_{\sharp s_i}\, u = \pi_i(M)\ \mathbf{in}\ u} \qquad \frac{\Gamma \vdash e : s_i \searrow M \qquad i \in \{1,2\}}{\Gamma \vdash \iota_i(e)_\ell : (s_1 + s_2, \ell) \searrow \iota_i(\mathbf{box}_{\sharp s_i}\, M)}$$

$$\frac{\Gamma \vdash e_0 : (s_1 + s_2, \ell) \searrow M_0 \qquad \Gamma, x : s_1 \vdash e_1 : s \searrow M_1 \qquad \Gamma, y : s_2 \vdash e_2 : s \searrow M_2}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ \iota_1(x) \Rightarrow e_1 \mid \iota_2(y) \Rightarrow e_2 : s \bullet \ell \searrow \mathbf{case}\ M_0\ \mathbf{of}\ \iota_1(x) \Rightarrow \mathbf{let\ box}_{\sharp s_1}\, u_x = x\ \mathbf{in}\ M_1 \mid \iota_2(y) \Rightarrow \mathbf{let\ box}_{\sharp s_2}\, u_y = y\ \mathbf{in}\ M_2}$$

Figure 4: Translation from SLam to $\lambda_s^\Box$

such undesirable increase of security levels. Unless a modal variable is really consumed, it can be passed to elsewhere by putting in a **box**, making the security level of the whole expression unrelated to that of the variable.

Correctness of the translation is given by Theorem 4.2.3. It requires auxiliary theorems stating that translation preserves typing and semantics with the following definitions.

Translation of SLam contexts to modal contexts is given as follows:

$$|x_1 : (t_1, \ell_1), \ldots, x_n : (t_n, \ell_n)| = u_{x_1} ::^{\ell_1} |t_1|, \ldots, u_{x_n} ::^{\ell_n} |t_n|$$

We write $e \Downarrow e'$ when $e \overset{*}{\rightsquigarrow} e'$ and there is no e'' such that $e' \longrightarrow e''$.

4.2.1 Theorem [Translation Preserves Typing]: If $\Gamma \vdash e : (t, \ell)$, then there exists M such that $\Gamma \vdash e : (t, \ell) \searrow M$ and $|\Gamma|; \cdot \vdash^\ell M : |t|$.

4.2.2 Theorem [Adequacy]: If $\vdash e : (t, \ell) \searrow M$ and t is ground, then $e \Downarrow e'$ iff $M \xrightarrow{*} M'$ and $\vdash e' : s \searrow M'$ for some $s \leq (t, \ell)$ and normal form M'.

4.2.3 Theorem [SLam Noninterference]: Let ℓ_1 and ℓ_2 be any two elements of $\mathcal{L}$. If $\ell_1 \not\sqsubseteq \ell_2$ and $x : (t, \ell_1) \vdash e : ((\mathit{unit}, \ell_2) + (\mathit{unit}, \ell_2), \ell_2)$, then for any e_1 and e_2 such that $\vdash e_i : (t, \ell_1)$, $[e_1/x]e \Downarrow v$ iff $[e_2/x]e \Downarrow v$.

Proof sketch: By Theorem 4.2.1, $x : (t, \ell_1) \vdash e : ((\mathit{unit}, \ell_2) + (\mathit{unit}, \ell_2), \ell_2) \searrow M$ and $u_x ::^{\ell_1}; \cdot \vdash M : \Box_{\ell_2}(\Box_{\ell_2}\mathit{unit} + \Box_{\ell_2}\mathit{unit})$. Then, the conclusion is immediate from Theorem 3.2.2. □

5 Related Work

Type-based Information Flow Analysis. As mentioned before, there have been a lot of type-based techniques of information flow analysis for various kinds of languages [10, 1, 23, 19, 3, 25]. (See also Sabelfeld and Myers [24] for an excellent survey of this area.) Among them, close to ours are of course ones for functional languages [10, 1, 23]. It is interesting to see that even their core type systems and semantics are slightly different from each

other and that the proofs of noninterference are also significantly different, accordingly. For example, Heintze and Riecke [10] and Abadi et al. [1] proved noninterference for variants of the SLam calculus, by using denotational techniques, while Pottier and Simonet [23] did it, by developing a customized operational semantics that can express two executions with different high security inputs at once. On the other hand, our noninterference proof is very simple, and essentially based on the observation that high security inputs simply disappear during reduction. Of course, this argument was possible as we have (1) full β-reduction, where any subterms—even terms under lambda abstractions—can be reduced and (2) nondeterministic reduction that allows to delay computation at a higher level.[1] So, we think it doesn't directly extend to a language with side-effects. Nevertheless, we believe it is worth noting that noninterference for a purely functional language *is* easy to prove.

To perform precise analysis for the while language with (first-order) procedures, Volpano and Smith [26] introduced procedures polymorphic with respect to security levels, that is, procedures parameterized by variables ranging over security levels. Although our calculus is not equipped with such a notion, it would be in principle possible to introduce the universal quantifier for possible worlds to our language. For example, the type of a function that takes two integers of the same security level and yields their sum might be written something like $\forall n \in \mathcal{L}.(\Box_n int \times \Box_n int \to \Box_n int)$.

Barthe and Serpette [4] developed a type system for information flow analysis (and binding time analysis) for $FOb_{1\leq:}$ [2], an object calculus with a first-order type system and subtyping, and proved noninterference. Their approach to proving noninterference is very similar to ours: both proofs are entirely syntactic and use the fact that a normal form at some level cannot contain higher-level variables. Our proof may appear slightly more involved since we use Church-Rosser and Strong Normalization properties, which are required only because we adopt full β-reduction and do not fix the evaluation strategy. On the other hand, Barthe and Serpette assumed the normal order for reduction; so, it always leads to a normal form (if any), making the proof look slightly simpler.

Monadic Type Systems and Lax Logic. One of the closest related work is Abadi et al.'s dependency core calculus (DCC) [1]. Its purpose is to give a unified account for more general program analysis—dependency analysis, of which information flow analysis is one instance. DCC, an extension of Moggi's computational lambda calculus [17], is equipped with monadic types, $T_\ell A$, indexed by a predetermined lattice element ℓ.

We believe that similarity to our modal types $\Box_\ell A$ is not superficial. In fact, the computational lambda calculus has been found to correspond to a modal logic called lax logic [5, 9]. One standard interpretation of lax modality, usually written $\bigcirc A$, is "A is true *under some constraint*," and the elimination rule of lax modality is given as:

$$\frac{\Gamma \vdash \bigcirc A \qquad \Gamma, A \vdash \bigcirc B}{\Gamma \vdash \bigcirc B}$$

It corresponds to the typing rule for monadic binding **bind** $x = M$ **in** N by interpreting $\bigcirc$ as the monadic type constructor. Later, Pfenning and Davies [21] pointed out that the lax modality $\bigcirc$ can be decomposed as "possibly necessary" $\Diamond\Box$. On the other hand, our modal types $\Box_\ell A$ could be decomposed as $@_\ell \Box A$, where $@_\ell A$ would mean "A is true at the world ℓ." In some sense, in $\lambda_s^\Box$, it is made explicit at which world $\Box A$ holds, while, in lax logic and the original computational lambda calculus, it is abstracted out by the possibility modality $\Diamond$. However, the typing rules of $\lambda_s^\Box$ and DCC are rather different. It is left for future work to figure out how they (do or do not) correspond to each other.

Type Systems Based On Modal Logic. Recently, several typed calculi based on proof systems of modal logic have been proposed for various purposes: staged computation [7], binding-time analysis in partial evaluation [6],

[1] Strong normalization and Church-Rosser guarantee that noninterference holds under any reduction strategy, though.

a formal account for the notion of meta-variables [20], and distributed computation [18, 13]. Each calculus (including $\lambda_s^{\Box}$) has slightly different modality, specialized to its purpose. To our knowledge, our work is the first to point out the relevance of modal logic to security or dependency analysis.

6 Conclusion

We have developed a typed lambda-calculus $\lambda_s^{\Box}$ to give a foundational account for type-based information flow analysis. The calculus corresponds, by (a natural extension of) the Curry-Howard isomorphism, to a proof system of an intuitionistic modal logic of local validity. The correspondence is based on the observation that security levels can be interpreted as possible worlds, legal directions of information flow can be as the reachability relation on possible worlds, and security types can be as propositions of validity. The calculus is shown to satisfy desirable properties including subject reduction, Church-Rosser, strong normalization. Moreover, we have found the noninterference property, a correctness property essential for information flow analysis, can be proved in a very simple syntactic manner, without involving denotational semantics or non-standard operational semantics. We also show that a purely functional core of the SLam calculus can be encoded into $\lambda_s^{\Box}$ and that its noninterference can be proved in terms of $\lambda_s^{\Box}$.

We briefly discuss possible future work below.

As mentioned in the last section, we conjecture that DCC's monadic types have strong connection with our modal types, although the type system is rather different. It is interesting work to investigate Curry-Howard isomorphism for DCC and study its logical interpretation.

We have studied a modal logic with validity as the only modality. It remains as an open question whether there is any sensible interpretation of modal possibility [21] in our context.

Recently, type-based information flow analysis for low-level languages such as the Java Virtual Machine Language (JVML) has been studied [16]. Since a type system for JVML can be interpreted as a variant of Gentzen's sequent calculus [14], we think it would be possible to apply our idea and develop a correspondence for low-level languages.

It may be an interesting question to answer how general our overall approach to foundations for type-based program analyses is. There have been a lot of type-based program analyses that use non-standard type systems—non-standard in the sense that types are decorated with information peculiar to the purpose of the analysis. It may be possible to find yet another correspondence between type-based program analyses and (modal) logic. Such work will be useful to deepen understanding of the essence of program analyses, as our work have been so for information flow analysis.

Acknowledgements. The authors are grateful to Masahiko Sato and anonymous reviewers for their useful comments. This work is supported in part by Grant-in-Aid for Scientific Research on Priority Areas Research No. 12133202 and Grant-in-Aid for Young Scientists (B) No. 15700011 from MEXT of Japan.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proc. of POPL '99*, pages 147–160, 1999.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[3] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in Java-like language. In *Proc. of 15th CSFW*, pages 253–267, 2002.

[4] Gilles Barthe and Bernard P. Serpette. Partial evaluation and non-interference for object calculi. In *Proc. of 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722 of *LNCS*, pages 53–67, Tsukuba, Japan, 1999.

[5] P. N. Benton, Gavin Bierman, and Valeria de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, 1998.

[6] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proc. of LICS'96*, pages 184–195, 1996.

[7] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[9] Matt Fairtlough and Michael Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997.

[10] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. of POPL '98*, pages 365–377, 1998.

[11] Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure information flows as typed process behaviour. In *Proc. of ESOP (ESOP)*, Springer LNCS 1782, pages 180–199, 2000.

[12] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proc. of 29th POPL (POPL'02)*, pages 81–92, Portland, OR, January 2002.

[13] Limin Jia and David Walker. Modal proofs as distributed programs. In *Proc. of ESOP*, volume 2986 of *LNCS*, pages 219–233, 2004.

[14] Shin-ya Katsumata and Atsushi Ohori. Proof-directed de-compilation of low-level code. In *Proc. of ESOP*, volume 2028 of *LNCS*, pages 352–366, 2001.

[15] Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. Technical Report TR03-0007, Dept. of Computer Science, Tokyo Institute of Technology, October 2003.

[16] Naoki Kobayashi and Keita Shirane. Type-based information flow analysis for JVML. In *Informal Proc. of APLAS'02*, 2002.

[17] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[18] Jonathan Moody. Modal logic as a basis for distributed computation. Technical Report CMU-CS-03-194, School of Computer Science, Carnegie Mellon University, 2003.

[19] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. of POPL'99*, pages 228–241, 1999.

[20] Aleksander Nanevski, Brigitte Pientka, and Frank Pfenning. A modal foundation for meta-variables. In *Proc. of Merλin*, 2003.

[21] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[22] François Pottier. A simple view of type-secure information flow in the pi-calculus. In *Proc. of CSFW*, pages 320–330, 2002.

[23] François Pottier and Vincent Simonet. Information flow inference in ML. *ACM TOPLAS*, 25(1):117–158, 2003.

[24] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal On Selected Areas In Communications*, 21(1):5–19, 2003.

[25] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-thereaded imperative language. In *Proc. of POPL*, pages 355–364, 1998.

[26] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proc. of TAPSOFT*, volume 1214 of *LNCS*, pages 607–621, 1997.

[27] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.