Noninterference in a Predicative Polymorphic Calculus for Access Control

Sepehr Amir-Mohammadian[a], Mehran S. Fallah[a]

[a] *Department of Computer Engineering and Information Technology, Amirkabir University of Technology (Tehran Polytechnic), P. O. Box: 15875-4413, Tehran, Iran*

Abstract

Polymorphic programming languages have been adapted for constructing distributed access control systems, where a program represents a proof of eligibility according to a given policy. As a security requirement, it is typically stated that the programs of such languages should satisfy noninterference. However, this property has not been defined and proven semantically. In this paper, we first propose a semantics based on Henkin models for a predicative polymorphic access control language based on lambda-calculus. A formal semantic definition of noninterference is then proposed through logical relations. We prove a type soundness theorem which states that any well-typed program of our language meets the noninterference property defined in this paper. In this way, it is guaranteed that access requests from an entity do not interfere with those from unrelated or more trusted entities.

Keywords: Access control, denotational semantics, noninterference, predicative polymorphism.

1. Introduction

The use of programming languages theory in providing information security has long been recognized [33, 34]. The basic idea is to design a programming language such that every system constructed using that language provably satisfies given security requirements. Although the research in this

Email addresses: samirmoh@uvm.edu (Sepehr Amir-Mohammadian), msfallah@aut.ac.ir (Mehran S. Fallah)

area has mainly focused on information confidentiality and integrity, the underlying notion can also be utilized in providing authenticity, e.g., [20, 13], service availability, e.g., [25, 42, 14], and even constructing secure access control systems, e.g., [16, 2, 15, 38, 35, 23]. When extended into access control, it may take the form of presenting an appropriate static semantics for the language so that its corresponding logic can reflect the intended policy. In this view, the programs of the language are regarded as proofs of eligibility and can be exploited in proof carrying authorization [8, 11].

A common approach to enforce confidentiality via programming languages theory is to build a variant of λ-calculus so that noninterference [19], which is the basic semantic notion of secure information flow, is satisfied by the terms of the calculus—the words calculus and language are used interchangeably throughout this paper. To achieve this, it is required to have a lattice of security labels representing the sensitivity of the information related to variables, terms, and even computations. The language is then so designed that information flow from high (private) to low (public) is prohibited by the type or run-time system. Integrity can be achieved in the same way where the lattice of labels represents the integrity of information, and then, flow from low (untrusted) to high (trusted) is prevented by the language.

A number of issues should be considered when noninterference is applied to access control. First, the language should be equipped with richer types handling specific relations between principals. For example, we need polymorphic types to formulate the assertions made about the relative power of principals such as those employed to encode the delegation of rights [2]. This necessitates a more complicated semantics, thereby complicating the study of noninterference. Second, the concept of noninterference, as it is defined in the realm of confidentiality, cannot be directly applied to the context of access control—we have seen similar results stating that if confidentiality and integrity are taken dual, some important facets will be ignored [27, 12].

If noninterference, as it is defined in confidentiality, is applied to a language that is employed as a logic for access control, different proofs of the same assertion made by an untrusted entity will have the same influence on what is derived from the policy. Such a property, however, vacuously holds in any logic, and thus, does reflect no specific feature of an access control system. This is while noninterference in the context of access control should guarantee that the policy will derive the same access rights in the presence of different assertions by an untrusted entity. By such an interpretation, a proof of an access right should not depend on any proof of any statement

by an untrusted entity. This is analogous to the interpretation of noninterference in some constructive logics of access control [17] where the proof theory of the logic is required to handle the assertions made by different principals independently of each other. An attempt has been made in [2] to bring this property into a calculus for access control based on Dependency Core Calculus (DCC) [4]—DCC is a λ-calculus devised to enforce restrictions on dependencies among computations through a lattice of monads [30, 40]. However, due to the lack of semantic treatment of the property, the soundness of the language was not provable. In [6], noninterference for a logic of access controls is studied in terms of the game semantics. This approach, however, is limited to the monomorphic calculi of access control. As indicated earlier, a more expressive language with richer types is required for access control.

This paper is an attempt to solve the problem stated above. In doing so, we first introduce $D^{\forall_P}$ which is a predicative polymorphic calculus based on [2]. Then, we propose a denotational semantics for the language using Henkin models [28] and employ it to introduce a noninterference property reflecting the features required for a secure access control system. As a type soundness theorem, we prove that every well-typed program of $D^{\forall_P}$ satisfies noninterference. Unlike impredicative polymorphic languages, a predicative one does not allow the application of terms to polymorphic types. Our language does not allow terms to be applied to terms of polymorphic types either. In this way, we are able to propose a straightforward semantics for the language, and consequently, to make the soundness of the language provable, but possibly at the cost of expressiveness. It is also discussed how such a language may make restrictions on what can be expressed as an access control policy.

It is worth noting that there are interesting works on languages with access control constructs, e.g., [10, 35, 24, 37], where a noninterference property can be defined and proven. In that line of research, access control policies are embedded in the code of programs and it is guaranteed that, for example, information flow from high to low does not occur when typable programs execute. In the line of research we follow, however, the language itself—and not a program—conveys what is required for access control. Put alternatively, the language acts as an authorization logic. In such a case, every well-typed program of the language can be thought of as a proof of eligibility. The noninterference property, then, states that derivable access rights do not depend on the statements made by untrusted entities.

The rest of this paper is organized as follows. Section 2 is on the significance of noninterference in access control systems. In Section 3, we develop

a predicative polymorphic calculus for access control named $D^{\forall_P}$. Section 4 gives a denotational semantics for the calculus based on Henkin models. Section 5 defines noninterference formally on the basis of the semantics of $D^{\forall_P}$. Then, it is shown that any well-typed term of the calculus satisfies noninterference. Section 6 is on the expressiveness of $D^{\forall_P}$ and Section 7 concludes the paper.

2. Noninterference and Access Control

Access control is, in general, the process of deciding whether or not a request for a resource should be granted. The decisions made by an access control system are based on the policy of the enterprise who intends to control access to its resources. Access control seems straightforward, though it is error prone, especially in distributed and open environments where there are diverse kinds of principals that perform a wide variety of operations. A principal may be a client, a node on the network, a communication link, or even a cryptographic key. Every principal may make requests for resources, delegate its own rights to the other principals, confirm the credentials provided by the others, and so on.

To manage complexities, modal logics have been proposed as a means for the specification and enforcement of access control policies, e.g., [5, 8, 26, 1, 16, 17, 2, 15, 23, 21, 18]. The use of modal operators such as *says* helps us abstract away the assertions made by principals from technical details concerning the manner of processing a request, authenticating the requester, and so on. In fact, a statement like $A\ says\ \sigma$ indicates that the principal A has made the assertion σ. The machinery of the logic can then be utilized to reason about access requests efficiently in such a way that the resources used for authorization can be minimized. In addition, the use of logic makes it possible to verify an access control system formally.

In a logical view, access control lists can be represented by formulas like $A\ controls\ \sigma$ which is a syntactic sugar for $(A\ says\ \sigma) \rightarrow \sigma$. A partial order relation, e.g., $speaksfor$, is also defined on the set of principals which specifies if a principal is at least as powerful as the other. By such an interpretation, $A\ speaks\ for\ B$ implies that we can employ every assertion made by A to validate B's assertions. It is typical to express $speaks\ for$ in terms of *says* as follows:

$$A\ speaks\ for\ B = \forall\sigma.\ A\ says\ \sigma \rightarrow B\ says\ \sigma.$$

One interesting way of implementing an access control system using logic is to build a programming language whose static semantics conveys the logical requirements for access control. In fact, inspired from the Curry-Howard isomorphism, authorization rules are represented by the typing rules of the language. In this view, an access request is granted if it can be derived in the logic. Equivalently, it is granted if, in the side of the corresponding programming language, there is a program of the type of what is requested. Thus, the principal having a request for a resource should provide the proof of its eligibility in the form of a program. The enterprise who authorizes principals then verifies the proof by type-checking that program. This approach has many advantages. First, the semantics of the logic can be studied through the massive work already done on the semantics of programming languages. Second, to implement an access control system, we should only check the proofs and it is not required to run a costly proof search.

The correctness of an access control system constructed in this way can be formulated as satisfying a noninterference property by all programs of the language. Noninterference, in general, states that high-level information should not be revealed in lower levels. This notion of security has been incorporated into a number of languages, e.g., [39, 22, 9, 31, 41, 43]. When used in the context of access control, an intuitive interpretation of noninterference is that granting an access to a principal must not be influenced by the assertions made by less trustworthy or unrelated principals. More formally, if A is more trustworthy than B, any proof M of an A's assertion, which relies on a proof N of an assertion made by B, can be converted to a proof being independent of N [2, 17].

The following example shows how the absence of this property may lead to an undesirable behavior of an access control system. Assume that principal A is more trusted than B and that

$$A \; controls \; \sigma, \tag{1}$$

where

$$\sigma = (B \; says \; \textsc{read}[\mathsf{fileX}]) \rightarrow \textsc{read}[\mathsf{fileX}].$$

Now, let M be a proof term of the statement $A \; says \; \sigma$. Furthermore, consider the case in which M can be obtained from the proof term N of

$$B \; says \; \textsc{read}[\mathsf{fileX}]. \tag{2}$$

In such a case, the statement $\textsc{read}[\mathsf{fileX}]$ can be derived when B issues the request represented by (2), i.e., there is a proof term N for (2). In fact, (2)

implies A *says* σ, due to the assumption that M is obtained from N. Then, from (1) and A *says* σ, we have (B *says* READ[fileX]) $\rightarrow$ READ[fileX]. This implies READ[fileX]. In this way, a less trustworthy principal B interferes with the decisions made by the access control system through the rights of a more trusted principal A. Such malfunctions will be avoided if an access control system enforces noninterference. In fact, noninterference is not satisfied in this example because M is obtained from N. That is, a less trusted principal interferes with the proofs of the assertions made by a more trusted one.

3. A Predicative Polymorphic Calculus for Access Control

We present $D^{\forall_P}$, a calculus that can be employed as a means for the specification and enforcement of access control policies. Indeed, it is a monadic predicative polymorphic language whose semantics enables us to define a noninterference property reflecting what is required for access control. In general, a type in a polymorphic language may have quantified type variables. The language is predicative because type application on quantified types is not allowed. Moreover, terms cannot be applied to terms of universal types. To study such languages, it is conventional to define small and large universes of types. The small universe, denoted by U_1, comprises the types with no quantified type variables, while the large universe, written U_2, consists of all types. The syntax of $D^{\forall_P}$, as given in Figure 1, defines the types and terms of the language in which t is a type variable, b is a base type, and A is a principal.

Figure 2 defines the judgment $\Gamma \vdash \tau : u$ which states that the type expression τ is in the universe u of types. In this figure, $\mathcal{B}$ is the set of base types and $\mathcal{L}$ is a lattice of principals in which $A \preceq B$ means that A is more trusted than B. The rule univ does not introduce a new type expression but indicates that the large universe subsumes the small one. It is also worth mentioning that a context Γ comprises a number of type variables associated with their universes as well as term variables paired off with their types. Moreover, in the rules defining $\Gamma \vdash \tau : u$ it is assumed that the context is well-formed—for the sake of conciseness, we do not add this assumption to the premises of the rules. The well-formedness of Γ is shown by $\diamond \vdash \Gamma$ and is defined in Figure 3. The rule wfc2 states that well-formedness is closed under adding a fresh type variable of the small universe—the freshness of t is shown by $t \# \Gamma$. The same holds for a fresh term variable x, as seen in the rule wfc3.

$$
\begin{array}{rcll}
\tau & ::= & t \mid b \mid \tau \to \tau \mid \tau * \tau \mid \tau + \tau \mid A \ says \ \tau \mid \forall t : U_1.\ \tau & Types \\
M & ::= & \langle\rangle \mid \mathbf{zero}^{\tau} \mid x \mid \lambda x : \tau.\ M \mid MM \mid \langle M, M \rangle \mid \mathbf{prj}_i\ M \mid \mathbf{inj}_i\ M & Terms \\
 & & \mid \mathbf{case}^{\tau_1,\tau_2,\tau_3} MMM \mid \mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ M \mid \eta_A M \mid \Lambda t : U_1.\ M \mid M\tau &
\end{array}
$$

Figure 1: Syntax of $D^{\forall_P}$.

$$
\begin{array}{lll}
\text{[base]} & \Gamma \vdash b : U_1 \quad (b \in \mathcal{B}) & \text{[t-var]} \quad \Gamma,\ t : U_1 \vdash t : U_1 \\
\text{[func]} & \dfrac{\Gamma \vdash \tau_1 : U_1 \quad \Gamma \vdash \tau_2 : U_1}{\Gamma \vdash \tau_1 \to \tau_2 : U_1} & \text{[prod]} \quad \dfrac{\Gamma \vdash \tau_1 : U_i \quad \Gamma \vdash \tau_2 : U_i \quad i \in \{1,2\}}{\Gamma \vdash \tau_1 * \tau_2 : U_i} \\
\text{[sum]} & \dfrac{\Gamma \vdash \tau_1 : U_i \quad \Gamma \vdash \tau_2 : U_i \quad i \in \{1,2\}}{\Gamma \vdash \tau_1 + \tau_2 : U_i} & \text{[says]} \quad \dfrac{\Gamma \vdash \tau : U_i \quad A \in \mathcal{L} \quad i \in \{1,2\}}{\Gamma \vdash A \ says \ \tau : U_i} \\
\text{[univ]} & \dfrac{\Gamma \vdash \tau : U_1}{\Gamma \vdash \tau : U_2} & \text{[poly]} \quad \dfrac{\Gamma,\ t : U_1 \vdash \tau : U_2}{\Gamma \vdash \forall t : U_1.\ \tau : U_2}
\end{array}
$$

Figure 2: Type expressions and their corresponding universes.

$$
\begin{array}{lll}
\text{[wfc1]} \quad \dfrac{}{\diamond \vdash \emptyset} & \text{[wfc2]} \quad \dfrac{\diamond \vdash \Gamma \qquad t \ \# \ \Gamma}{\diamond \vdash \Gamma,\ t : U_1} & \text{[wfc3]} \quad \dfrac{\Gamma \vdash \tau : U_i \qquad x \ \# \ \Gamma}{\diamond \vdash \Gamma,\ x : \tau}
\end{array}
$$

Figure 3: Well-formed contexts.

$$[\mathsf{pr1}] \quad \pi(\mathit{unit},\ A) \qquad [\mathsf{pr2}] \quad \pi(\mathit{null} \to \tau,\ A)$$

$$[\mathsf{pr3}] \quad \frac{A \preceq B}{\pi(B\ \mathit{says}\ \tau,\ A)} \qquad [\mathsf{pr4}] \quad \frac{\pi(\tau,\ A)}{\pi(B\ \mathit{says}\ \tau,\ A)}$$

$$[\mathsf{pr5}] \quad \frac{\pi(\tau_1,\ A) \qquad \pi(\tau_2,\ A)}{\pi(\tau_1 * \tau_2,\ A)} \qquad [\mathsf{pr6}] \quad \frac{\pi(\tau_2,\ A)}{\pi(\tau_1 \to \tau_2,\ A)}$$

$$[\mathsf{pr7}] \quad \frac{\forall \tau_1 : U_1\,.\,\pi([\tau_1/t]\tau_2, A)}{\pi(\forall t : U_1.\tau_2, A)}$$

Figure 4: Protection rules.

We also need to define protection at a given trust level. We may use the judgment $\pi(\tau,\ A)$, defined in Figure 4, to state that the type τ is protected at level A. When used in access control, each type is matched with a statement of the corresponding logic—the Curry-Howard isomorphism. Thus, protection can be interpreted in terms of the influence of a statement on a particular level. More precisely, $\pi(\tau,\ A)$ ensures that the statement τ will not influence the proof of the statements representing privileges of A. This is akin to what is required for information integrity.

In $D^{\forall_P}$, we have two distinguished base types *unit* and *null*. The base type *unit* is equivalent to $\top$ in the side of logic, whereas *null* represents falsehood $\bot$. Thus, *unit* does not influence other proofs; it holds anywhere vacuously. The same holds for the type $\mathit{null} \to \tau$. Other base types correspond to the statements representing the final decisions made by an access control system, e.g., "fileX is deleted." Such base types may act as hypotheses for the proof of other statements. Therefore, they are considered to be protected at no level. The operator *says* is a type constructor that takes an element A of lattice $\mathcal{L}$ and a type and returns a specific type that is protected at levels less than A. If principal B is less trusted than A, then what B says should not influence the proof of statements representing access to the objects controlled by A. In the case where principal B is not less trusted than A, B *says* τ is protected at level A if τ is protected at the same. A product type is protected at a level if both components are protected at that level. A function type is protected as long as the return type is. Finally, a polymorphic type is protected at a particular level if every type obtained from replacing the type variable with a specific type is protected. Note that sum types are not protected at any level, as they can influence what is more trusted through injection tags [36].

[unit] $\Gamma \vdash \langle\,\rangle : unit$

[zero] $\Gamma \vdash \mathbf{zero}^{\tau} : null \to \tau$

[var] $\Gamma,\ x : \tau \vdash x : \tau$

[abst] $$\frac{\Gamma,\ x : \tau_1 \vdash M : \tau_2 \qquad \Gamma \vdash \tau_1 : U_1 \qquad \Gamma \vdash \tau_2 : U_1}{\Gamma \vdash \lambda x : \tau_1.\ M : \tau_1 \to \tau_2}$$

[app] $$\frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}$$

[pair] $$\frac{\Gamma \vdash M : \tau_1 \qquad \Gamma \vdash N : \tau_2}{\Gamma \vdash \langle M, N\rangle : \tau_1 * \tau_2}$$

[proj-i] $$\frac{\Gamma \vdash M : \tau_1 * \tau_2 \qquad i \in \{1,2\}}{\Gamma \vdash \mathbf{prj}_i\ M : \tau_i}$$

[inj-i] $$\frac{\Gamma \vdash M : \tau_i \qquad i \in \{1,2\}}{\Gamma \vdash \mathbf{inj}_i\ M : \tau_1 + \tau_2}$$

[case] $$\frac{\Gamma \vdash M : \tau_1 + \tau_2 \qquad \Gamma \vdash N : \tau_1 \to \tau_3 \qquad \Gamma \vdash P : \tau_2 \to \tau_3}{\Gamma \vdash \mathbf{case}^{\tau_1,\tau_2,\tau_3} MNP : \tau_3}$$

[unitM] $$\frac{\Gamma \vdash M : \tau \qquad A \in \mathcal{L}}{\Gamma \vdash \boldsymbol{\eta}_A M : A\ says\ \tau}$$

[bindM] $$\frac{\Gamma \vdash M : A\ says\ \tau_1 \qquad \Gamma,\ x : \tau_1 \vdash N : \tau_2 \qquad \pi(\tau_2,\ A)}{\Gamma \vdash (\mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ N) : \tau_2}$$

[t-abst] $$\frac{\Gamma,\ t : U_1 \vdash M : \tau}{\Gamma \vdash (\Lambda t : U_1.\ M) : (\forall t : U_1.\ \tau)}$$

[t-app] $$\frac{\Gamma \vdash M : (\forall t : U_1.\ \tau_1) \qquad \Gamma \vdash \tau_2 : U_1}{\Gamma \vdash M\tau_2 : [\tau_2/t]\tau_1}$$

Figure 5: Typing rules.

The typing rules of $D^{\forall_P}$ are given in Figure 5. All rules other than unitM and bindM are standard typing rules for a predicative polymorphic λ-calculus. The rule unitM introduces the terms representing computations. It is the same as the standard typing rule for a monadic calculus except that the level of computation should be known in advance—$A \in \mathcal{L}$. The rule bindM is the binding rule for computations in which the protection judgment appears as a premise.

4. Semantics

In this section, we propose a denotational semantics for $D^{\forall_P}$ using Henkin models. It is based on classical set theory where a function type is interpreted as the set of all mathematical functions from a specific domain to the co-domain that is denoted by its return type.

The applicative structure for $D^{\forall_P}$ is defined to be the triple

$$\mathcal{A} = \langle \mathcal{U}, dom, \{\mathbf{App}^{a,b}, \mathbf{App}^{f}, \mathbf{Proj}_1^{a,b}, \mathbf{Proj}_2^{a,b}, \mathbf{Inleft}^{a,b}, \mathbf{Inright}^{a,b}, \mathbf{T}_{A,a}\}\rangle,$$

where

1. $\mathcal{U} = \left\{U_1^{\mathcal{A}}, U_2^{\mathcal{A}}, \rightarrow^{\mathcal{A}}, \forall^{\mathcal{A}}, *^{\mathcal{A}}, +^{\mathcal{A}}, \{T_A^{\mathcal{A}} | A \in \mathcal{L}\}, \mathcal{I}_{type}\right\}$ in which
   - $U_1^{\mathcal{A}}$ and $U_2^{\mathcal{A}}$ are two sets such that $U_1^{\mathcal{A}} \subseteq U_2^{\mathcal{A}}$ and $\mathbf{0}, \mathbf{1} \in U_1^{\mathcal{A}}$,
   - $\rightarrow^{\mathcal{A}}$: $U_1^{\mathcal{A}} \times U_1^{\mathcal{A}} \rightarrow U_1^{\mathcal{A}}$,
   - $\forall^{\mathcal{A}} : [U_1^{\mathcal{A}} \rightarrow U_2^{\mathcal{A}}] \rightarrow U_2^{\mathcal{A}}$, where $[U_1^{\mathcal{A}} \rightarrow U_2^{\mathcal{A}}] = \{f | f : U_1^{\mathcal{A}} \rightarrow U_2^{\mathcal{A}}\}$,
   - $*^{\mathcal{A}} : U_2^{\mathcal{A}} \times U_2^{\mathcal{A}} \rightarrow U_2^{\mathcal{A}}$,
   - $+^{\mathcal{A}} : U_2^{\mathcal{A}} \times U_2^{\mathcal{A}} \rightarrow U_2^{\mathcal{A}}$,
   - $T_A^{\mathcal{A}} : U_2^{\mathcal{A}} \rightarrow U_2^{\mathcal{A}}$, and
   - $\mathcal{I}_{type} : \mathcal{B} \rightarrow U_1^{\mathcal{A}}$ with $\mathcal{I}_{type}(null) = \mathbf{0}$ and $\mathcal{I}_{type}(unit) = \mathbf{1}$; $\mathcal{I}_{type}$ is an injective function as well.

   Throughout this paper, we will use $\rightarrow^{\mathcal{A}}$, $*^{\mathcal{A}}$, and $+^{\mathcal{A}}$ as infix operators.

2. $dom = \{Dom^a | a \in U_2^{\mathcal{A}}\}$, where Dom^a is a set of values for the type value $a \in U_2^{\mathcal{A}}$; $Dom^{\mathbf{0}} = \{\}$ and $Dom^{\mathbf{1}} = \{\top\}$, and

3. for $\{\mathbf{App}^{a,b}, \mathbf{App}^{f}, \mathbf{Proj}_1^{a,b}, \mathbf{Proj}_2^{a,b}, \mathbf{Inleft}^{a,b}, \mathbf{Inright}^{a,b}, \mathbf{T}_{A,a}\}$, we have
   - $\mathbf{App}^{a,b} : Dom^{a \rightarrow^{\mathcal{A}} b} \rightarrow Dom^a \rightarrow Dom^b$ for $a, b \in U_1^{\mathcal{A}}$,
   - $\mathbf{App}^{f} : Dom^{\forall^{\mathcal{A}}(f)} \rightarrow U_1^{\mathcal{A}} \rightarrow \bigcup_{a \in U_2^{\mathcal{A}}} Dom^a$, where for every value $v \in Dom^{\forall^{\mathcal{A}}(f)}$ and every type value $a \in U_1^{\mathcal{A}}$, we have $\mathbf{App}^f v\ a \in Dom^{f(a)}$,
   - $\mathbf{Proj}_1^{a,b} : Dom^{a *^{\mathcal{A}} b} \rightarrow Dom^a$ for $a, b \in U_2^{\mathcal{A}}$,
   - $\mathbf{Proj}_2^{a,b} : Dom^{a *^{\mathcal{A}} b} \rightarrow Dom^b$ for $a, b \in U_2^{\mathcal{A}}$,

- $\mathbf{Inleft}^{a,b} : Dom^a \to Dom^{a+^{\mathcal{A}}b}$ for $a, b \in U_2^{\mathcal{A}}$,
- $\mathbf{Inright}^{a,b} : Dom^b \to Dom^{a+^{\mathcal{A}}b}$ for $a, b \in U_2^{\mathcal{A}}$, and
- $\mathbf{T}_{A,a} : Dom^a \to Dom^{T_A^{\mathcal{A}}(a)}$ for $a \in U_2^{\mathcal{A}}$ and $A \in \mathcal{L}$.

The applicative structure $\mathcal{A}$ should be extensional. In addition to the conditions stated in [28], this requires the following extra condition.

$$\forall f, f' \in Dom^{T_A^{\mathcal{A}}(a) \to b} . \forall d \in Dom^a .$$
$$\mathbf{App}^{T_A^{\mathcal{A}}(a),b} f(\mathbf{T}_{A,a} d) = \mathbf{App}^{T_A^{\mathcal{A}}(a),b} f'(\mathbf{T}_{A,a} d) \Rightarrow f = f'.$$

Moreover, for the applicative structure $\mathcal{A}$, the $\mathcal{A}$-environment ν is defined by

$$\nu : Var \to U_1^{\mathcal{A}} \cup \bigcup_{a \in U_2^{\mathcal{A}}} Dom^a,$$

where Var is the set of variables. Then, for every type variable t, $\nu(t) \in U_1^{\mathcal{A}}$. We use $\nu[a/t]$ and $\nu[x \mapsto a]$ to denote environments in which a particular variable is mapped to a specific value. The environment $\nu[a/t]$ represents the state in which t is mapped to the type value a. More formally,

$$\nu[a/t](s) = \begin{cases} a, & s = t \\ \nu(s), & s \neq t. \end{cases}$$

Similarly, the environment $\nu[x \mapsto a]$ represents the state in which x is mapped to the term value a.

Given an applicative structure $\mathcal{A}$ and an $\mathcal{A}$-environment ν, the meaning of a type expression τ, written $[\![\tau]\!]_\nu$, is defined by

- $[\![t]\!]_\nu = \nu(t) \ \ (t \in Var)$,
- $[\![b]\!]_\nu = \mathcal{I}_{type}(b) \ \ (b \in \mathcal{B})$,
- $[\![\tau \to \tau']\!]_\nu = [\![\tau]\!]_\nu \to^{\mathcal{A}} [\![\tau']\!]_\nu$,
- $[\![\tau * \tau']\!]_\nu = [\![\tau]\!]_\nu *^{\mathcal{A}} [\![\tau']\!]_\nu$,
- $[\![\tau + \tau']\!]_\nu = [\![\tau]\!]_\nu +^{\mathcal{A}} [\![\tau']\!]_\nu$,
- $[\![A \ says \ \tau]\!]_\nu = T_A^{\mathcal{A}}([\![\tau]\!]_\nu)$, and

- $[\![\forall t : U_1.\ \tau]\!]_\nu = \forall^{\mathcal{A}}(f)$ where $f : U_1^{\mathcal{A}} \to U_2^{\mathcal{A}}$ and $f(a) = [\![\tau]\!]_{\nu[a/t]}$.

An environment ν satisfies a context Γ, written $\nu \models \Gamma$, iff

$$x : \tau \in \Gamma \Rightarrow \nu(x) \in Dom^{[\![\tau]\!]_\nu}.$$

We follow the meaning of term M of type τ in context Γ and context-satisfying environment ν as defined in [28] and give meaning to the new terms of $D^{\forall_P}$ by

$$[\![\Gamma \vdash \boldsymbol{\eta}_A M : A\ says\ \tau]\!]_\nu = \mathbf{T}_{A,a}[\![\Gamma \vdash M : \tau]\!]_\nu, \tag{3}$$

and

$$[\![\Gamma \vdash (\mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ N) : \tau']\!]_\nu = [\![\Gamma,\ x : \tau \vdash N : \tau']\!]_{\nu[x \mapsto d]} \tag{4}$$

in which $a = [\![\tau]\!]_\nu$, $d \in Dom^a$, and $[\![\Gamma \vdash M : A\ says\ \tau]\!]_\nu = \mathbf{T}_{A,a}d$.

An extensional applicative structure $\mathcal{A}$ is a Henkin model if the meaning function "$[\![\cdot]\!]_\cdot$" defined above is a total function.

Lemma 1. *For any two type expressions τ and τ', any type variable t, and any environment ν, $[\![[\tau'/t]\tau]\!]_\nu = [\![\tau]\!]_{\nu[[\![\tau']\!]_\nu/t]}$.*

Proof. By induction on the structure of τ. □

Theorem 1. *The meaning function "$[\![\cdot]\!]_\cdot$" is total.*

Proof. By induction on the rules defining $\Gamma \vdash M : \tau$. □

5. Noninterference and Type Soundness

Noninterference is of great concern in distributed access control systems, where there exist diverse kinds of requesting entities, access rights, and requests. In such environments, it is significant to have trust in what is asserted by a principal and the way an access permission is derived. A principal can influence the derivation of an access permission through asserting those statement formulated by *says* constructor in $D^{\forall_P}$. Noninterference can then be defined as a property requiring that untrusted principals may not influence the derivation of access permissions for trusted principals.

To formalize this notion, we first consider the family of semantic relations

$$\mathcal{R} = \{R_A^{a,a'} \subseteq Dom^a \times Dom^{a'} | a, a' \in U_2^{\mathcal{A}},\ A \in \mathcal{L}\}$$

which primarily represent the extent to which a principal trusts a computation or equivalently a proof. Rather than being indexed by a single type value, which is usual in logical relations as well as in defining and analyzing information flow properties, e.g., [22, 4], the relations here are defined over two type values. This will facilitate the semantic interpretation of noninterference in access control. In access control, we need to say that replacing some statement τ from a principal with any other statement σ from the same does not influence the proof of the statement originated from a more trustworthy principal. Such a property cannot be expressed using semantic relations on single type denotations where the proof of a statement can only be substituted for another proof of the same statement. In fact, making restrictions on replacing a proof of a statement with another proof of the same statement does not reflect what is expected from noninterference in the context of access control.

Our semantic relations are inductively defined in Table 1. If $(v, v') \in R_A^{a,a'}$, then, in principal A's view, $v \in Dom^a$ is denoted by a proof term that can be replaced with another proof term denoting $v' \in Dom^{a'}$. We explain the relation $R_A^{a,a'}$ through an example. Consider the type value $a \in U_2^{\mathcal{A}}$. If principal A trusts a, $R_A^{a,a'} = \emptyset$ holds for any type value $a' \in U_2^{\mathcal{A}}$ other than a. Put alternatively, in the side of the corresponding logic, A does discriminate between a proof of a statement (type) with denotation a and a proof of a statement with another denotation. However, if A does not trust a, then $R_A^{a,a'} = Dom^a \times Dom^{a'}$ holds for any type value a' that is not trustworthy to A. That is, A is indifferent between the proofs of the statement denoting a and the ones of any untrusted statement denoting a'. From now on, we say that the relation $R_A^{a,a'}$ is complete if $R_A^{a,a'} = Dom^a \times Dom^{a'}$. Another case of indifference for A would be when there exist two proofs for a single statement, i.e., when $a = a'$. Indeed, what matters for the principal A is the provability of the statement rather than what proof is associated with the statement. Therefore, $R_A^{a,a}$ would be complete as well.

The denotations of base types are assumed to be trustworthy to any principal. Thus, for base types $b, b' \in \mathcal{B}$ with $[\![b]\!]_\nu \neq [\![b']\!]_{\nu'}$ the relation $R_A^{[\![b]\!]_\nu,[\![b']\!]_{\nu'}}$ is empty. The relation for the base types of the same denotation is complete. It should be noted that since the function $\mathcal{I}_{type}$ is injective, the base types of the same denotation are the same. The relation concerning the denotations of two function types comprises all pairs (f, f') that, in a principal A's vision, preserve closure with respect to $\mathcal{R}$. That is, their return

Table 1: Definition of $R_A^{a,a'}$.

$$R_A^{[\![b]\!]_\nu,[\![b']\!]_{\nu'}} = \begin{cases} \emptyset, & b \neq b' \\ Dom^{[\![b]\!]_\nu} \times Dom^{[\![b']\!]_{\nu'}}, & b = b' \end{cases} \qquad (b, b' \in \mathcal{B})$$

$$R_A^{[\![\tau\to\tau']\!]_\nu,[\![\sigma\to\sigma']\!]_{\nu'}} =$$
$$\Big\{(f,f') \mid f \in Dom^{[\![\tau\to\tau']\!]_\nu} \wedge f' \in Dom^{[\![\sigma\to\sigma']\!]_{\nu'}} \wedge$$
$$\forall (a,a') \in R_A^{[\![\tau]\!]_\nu,[\![\sigma]\!]_{\nu'}} . (\mathbf{App}^{[\![\tau]\!]_\nu,[\![\tau']\!]_\nu} f a, \mathbf{App}^{[\![\sigma]\!]_{\nu'},[\![\sigma']\!]_{\nu'}} f' a') \in R_A^{[\![\tau']\!]_\nu,[\![\sigma']\!]_{\nu'}} \Big\}$$

$$R_A^{[\![\tau*\tau']\!]_\nu,[\![\sigma*\sigma']\!]_{\nu'} =}$$
$$\Big\{(p,p') \mid p \in Dom^{[\![\tau*\tau']\!]_\nu} \wedge p' \in Dom^{[\![\sigma*\sigma']\!]_{\nu'}} \wedge (\mathbf{Proj}_1^{[\![\tau]\!]_\nu,[\![\tau']\!]_\nu} p, \mathbf{Proj}_1^{[\![\sigma]\!]_{\nu'},[\![\sigma']\!]_{\nu'}} p') \in R_A^{[\![\tau]\!]_\nu,[\![\sigma]\!]_{\nu'}} \wedge$$
$$(\mathbf{Proj}_2^{[\![\tau]\!]_\nu,[\![\tau']\!]_\nu} p, \mathbf{Proj}_2^{[\![\sigma]\!]_{\nu'},[\![\sigma']\!]_{\nu'}} p') \in R_A^{[\![\tau']\!]_\nu,[\![\sigma']\!]_{\nu'}} \Big\}$$

$$R_A^{[\![\tau+\tau']\!]_\nu,[\![\sigma+\sigma']\!]_{\nu'} =}$$
$$\Big\{(\mathbf{Inleft}^{[\![\tau]\!]_\nu,[\![\tau']\!]_\nu} a, \mathbf{Inleft}^{[\![\sigma]\!]_{\nu'},[\![\sigma']\!]_{\nu'}} a') \mid a \in Dom^{[\![\tau]\!]_\nu} \wedge a' \in Dom^{[\![\sigma]\!]_{\nu'}} \wedge (a,a') \in R_A^{[\![\tau]\!]_\nu,[\![\sigma]\!]_{\nu'}} \Big\} \cup$$
$$\Big\{(\mathbf{Inright}^{[\![\tau]\!]_\nu,[\![\tau']\!]_\nu} a, \mathbf{Inright}^{[\![\sigma]\!]_{\nu'},[\![\sigma']\!]_{\nu'}} a') \mid a \in Dom^{[\![\tau']\!]_\nu} \wedge a' \in Dom^{[\![\sigma']\!]_{\nu'}} \wedge (a,a') \in R_A^{[\![\tau']\!]_\nu,[\![\sigma']\!]_{\nu'}} \Big\}$$

$$R_A^{[\![B \text{ says } \tau]\!]_\nu,[\![B \text{ says } \sigma]\!]_{\nu'}} = \begin{cases} \Big\{(\mathbf{T}_{B,[\![\tau]\!]_\nu} a, \mathbf{T}_{B,[\![\sigma]\!]_{\nu'}} a') \mid (a,a') \in R_A^{[\![\tau]\!]_\nu,[\![\sigma]\!]_{\nu'}} \Big\}, & B \preceq A \\ Dom^{[\![B \text{ says } \tau]\!]_\nu} \times Dom^{[\![B \text{ says } \sigma]\!]_{\nu'}}, & B \not\preceq A \end{cases}$$

$$R_A^{[\![\forall t:U_1.\ \tau]\!]_\nu,[\![\forall t:U_1.\ \sigma]\!]_{\nu'} =}$$
$$\Big\{(f,f') \mid f \in Dom^{[\![\forall t:U_1.\ \tau]\!]_\nu} \wedge f' \in Dom^{[\![\forall t:U_1.\ \sigma]\!]_{\nu'}} \wedge \forall F, F' : U_1^{\mathcal{A}} \to U_2^{\mathcal{A}} . \forall \tau', \sigma' \in U_1 .$$
$$\Big(F(z) = [\![\tau]\!]_{\nu[z/t]} \wedge F'(z) = [\![\sigma]\!]_{\nu'[z/t]}\Big) \to$$
$$\Big(\mathbf{App}^F f [\![\tau']\!]_\nu, \mathbf{App}^{F'} f' [\![\sigma']\!]_{\nu'}\Big) \in R_A^{[\![\tau]\!]_{\nu[[\![\tau']\!]_\nu/t]},[\![\sigma]\!]_{\nu'[[\![\sigma']\!]_{\nu'}/t]}} \Big\}$$

$$R_A^{[\![\tau]\!]_\nu,[\![\sigma]\!]_{\nu'}} = \emptyset, \qquad \textit{otherwise.}$$

values are related for any pair of related input elements. In fact, in A's viewpoint, the λ-term denoting f can be replaced with the term denoting f'. For the denotations of product types, p is in relation with p' if the terms denoting the components of p can be replaced with the terms denoting the components of p'. The denotations of two sum types are related if they are the left or right injection of related denotations of respective types.

Principal A trusts the computations of type B *says* τ depending on B's trustworthiness in comparison with A's. If A is less trustworthy than B, from A's viewpoint, the denotations of two computations at level B are related provided the denotations of the terms on which computations are performed are related. If A is not less trustworthy than B, any pair of values denoted by computations at level B are related. Finally, the denotations of two polymorphic types are in relation with each other in a principal's vision if the results of their application to the denotation of every type are related.

Theorem 2. *Assume that* $\Gamma \vdash M : \sigma$ *and* $\nu, \nu' \models \Gamma$ *for two environments* ν *and* ν'*. If* $\big(\nu(x), \nu'(x)\big) \in R_A^{[\![\tau]\!]_\nu, [\![\tau]\!]_{\nu'}}$ *for all* $x : \tau \in \Gamma$*, then*

$$\big([\![\Gamma \vdash M : \sigma]\!]_\nu, [\![\Gamma \vdash M : \sigma]\!]_{\nu'}\big) \in R_A^{[\![\sigma]\!]_\nu, [\![\sigma]\!]_{\nu'}}.$$

Proof. By induction on the typing rules given in Figure 5. Here, we sketch the proof under the derivation by the rule t-app. In doing so, we assume that the theorem holds for $\Gamma \vdash M : \forall t : U_1 \,.\, \tau_1$ and then we prove it for $\Gamma \vdash M\tau_2 : [\tau_2/t]\tau_1$. If $\Gamma \vdash M\tau_2 : [\tau_2/t]\tau_1$, $\nu, \nu' \models \Gamma$, and for all $x : \tau \in \Gamma$, $\big(\nu(x), \nu'(x)\big) \in R_A^{[\![\tau]\!]_\nu, [\![\tau]\!]_{\nu'}}$, then according to the induction hypothesis we have

$$\big([\![\Gamma \vdash M : \forall t : U_1 \,.\, \tau_1]\!]_\nu, [\![\Gamma \vdash M : \forall t : U_1 \,.\, \tau_1]\!]_{\nu'}\big) \in R_A^{[\![\forall t:U_1 \,.\, \tau_1]\!]_\nu, [\![\forall t:U_1 \,.\, \tau_1]\!]_{\nu'}}.$$

Then, according to the definition of our logical relation over polymorphic types in Table 1,

$$\Big(\mathbf{App}^F [\![\Gamma \vdash M : \forall t : U_1 \,.\, \tau_1]\!]_\nu a, \\ \mathbf{App}^{F'} [\![\Gamma \vdash M : \forall t : U_1 \,.\, \tau_1]\!]_{\nu'} a\Big) \in R_A^{[\![\tau_1]\!]_{\nu[a/t]}, [\![\tau_1]\!]_{\nu'[a/t]}}$$

holds for all $a \in U_1^{\mathcal{A}}$, where F and F' denote the functions that are used to model the type $\forall t : U_1 \,.\, \tau_1$ in ν and ν', respectively. Therefore, the above pair could be formulated as the pair of denotations for $\Gamma \vdash M\tau_2 : [\tau_2/t]\tau_1$ in ν and ν'. □

Now, we define noninterference in the context of access control as a property to which principals cannot interfere with those requests made by more trustworthy or unrelated principals. In doing so, first we define $eval_A$ as follows:

$$eval_A \stackrel{def}{=} \Lambda t : U_1 \,.\, \lambda z : A \; says \; t \,.\, \mathbf{let}\; y \Leftarrow z \;\mathbf{in}\; y.$$

The term $eval_A$ takes a type and a computation of that type in level A and returns the result of the computation. It is worth noting that $eval_A$ can only be applied to those types that are protected at level A.

Definition 1. A term N of type σ in context Γ, $t : U_1$, $x : t$ satisfies noninterference iff for every two type expressions τ and τ' and every principal

A for which $[\tau/t]\sigma$ and $[\tau'/t]\sigma$ are not protected at A and every two terms M and M' of types A *says* τ and A *says* τ',

$$\Big(\llbracket \Gamma \vdash N'\,\tau\,(eval_A\ \tau\ M) : [\tau/t]\sigma \rrbracket_\nu,$$
$$\llbracket \Gamma \vdash N'\,\tau'\,(eval_A\ \tau'\ M') : [\tau'/t]\sigma \rrbracket_{\nu'}\Big) \in R_C^{\llbracket [\tau/t]\sigma \rrbracket_\nu, \llbracket [\tau'/t]\sigma \rrbracket_{\nu'}}$$

holds for every two context-satisfying environments ν and ν' and every principal C, where $[\tau/t]\sigma$ and $[\tau'/t]\sigma$ are protected at C and

$$N' = \big(\Lambda t : U_1\,.\,\lambda x : t\,.\,N\big)\ :\ \big(\forall t : U_1\,.\,t \to \sigma\big).$$

The definition also implicitely assumes that $eval_A\ \tau\ M$ and $eval_A\ \tau'\ M'$ are well-typed.

In a colloquial sense, a term N of type σ with free variable x satisfies noninterference if, in every principal C's view, it does not matter which untrusted computation is substituted for x. Its logical interpretation is as follows: if A is less trustworthy than C, the proof of σ does not depend on a proof M of A *says* τ, and thus, M can be replaced with any proof M' of any other statement A *says* τ'.

In should be noted that our definition of a term satisfying noninterference is reminiscent of Reynolds' relational parametricity [32]. It states that the clients of an abstract data type behave uniformly for all available interpretations of that type and cannot depend on the way it is represented. A useful application of relational parametricity is the notion of representation independence [29, 7] which states that two different implementations of an abstract type are equivalent if there is a relation between their type representations that is preserved by their operations. The semantics handling such a relation between types may be represented by logical relations over pair of type expressions.

Definition 2. We say that a language enforces noninterference, or is type-sound, iff every well-typed term of that language satisfies noninterference.

Theorem 3. *$D^{\forall_P}$ is type-sound.*

Proof. There are two kinds of typable terms, those protected at some level and those that are not protected at any level. For the terms of the second

kind, the proof is immediate, as the premise of the theorem does not hold. For the terms that are protected at some level, we can use induction on the protection rules given in Figure 4. More precisely, it is proven that the property P, defined below, holds of $\pi([\tau/t]\sigma,\ C)$.

$$
\begin{aligned}
&P(\pi([\tau/t]\sigma,\ C)) = \\
&\Big(\Gamma,\ t : U_1,\ x : t \vdash N : \sigma\ \wedge \neg\pi([\tau/t]\sigma,\ A)\ \wedge \neg\pi([\tau'/t]\sigma,\ A) \wedge \\
&\Gamma \vdash M : A\ says\ \tau\ \wedge \Gamma \vdash M' : A\ says\ \tau'\ \wedge \pi([\tau/t]\sigma,\ C)\ \wedge \pi([\tau'/t]\sigma,\ C) \wedge \\
&\Gamma \vdash eval_A\ \tau\ M : \tau\ \wedge\ \Gamma \vdash eval_A\ \tau'\ M' : \tau'\Big) \Rightarrow \\
&\Big(\llbracket \Gamma \vdash (\Lambda t : U_1\,.\,\lambda x : t\,.\,N)\,\tau\,(eval_A\ \tau\ M) : [\tau/t]\sigma \rrbracket_\nu, \\
&\quad \llbracket \Gamma \vdash (\Lambda t : U_1\,.\,\lambda x : t\,.\,N)\,\tau'\,(eval_A\ \tau'\ M') : [\tau'/t]\sigma \rrbracket_{\nu'}\Big) \in R_C^{\llbracket [\tau/t]\sigma \rrbracket_\nu, \llbracket [\tau'/t]\sigma \rrbracket_{\nu'}}.
\end{aligned}
$$

By $\neg\pi(\sigma,\ A)$, we mean σ is not protected at level A. We prove the property over $\pi(B\ says\ [\tau/t]\sigma,\ C)$ here—other cases can be treated similarly. In doing so, we use induction on the rules pr3 and pr4.

- **Proof under derivation by the rule pr3**: We show that if $C \preceq B$ then $P(\pi(B\ says\ [\tau/t]\sigma,\ C))$. The type $B\ says\ [\tau/t]\sigma$ is not protected at level A. Therefore, according to pr3 and pr4, $A \not\preceq B$ and $[\tau/t]\sigma$ is not protected at level A. Similarly, from $\neg\pi(B\ says\ [\tau'/t]\sigma,\ A)$, it is concluded that $[\tau'/t]\sigma$ is not protected at level A. From $C \preceq B$ and $A \not\preceq B$, we have $A \not\preceq C$. Now, we define d and d' as $d = \llbracket \Gamma \vdash M : A\ says\ \tau \rrbracket_\nu$ and $d' = \llbracket \Gamma \vdash M' : A\ says\ \tau' \rrbracket_{\nu'}$. From $A \not\preceq C$, we have

$$R_C^{\llbracket A\ says\ \tau \rrbracket_\nu, \llbracket A\ says\ \tau' \rrbracket_{\nu'}} = Dom^{\llbracket A\ says\ \tau \rrbracket_\nu} \times Dom^{\llbracket A\ says\ \tau' \rrbracket_{\nu'}}.$$

  Hence,

$$(d, d') \in R_C^{\llbracket A\ says\ \tau \rrbracket_\nu, \llbracket A\ says\ \tau' \rrbracket_{\nu'}}. \tag{5}$$

  We also define f and f' as

$$
\begin{aligned}
f = \Big\llbracket &\Gamma,\ t : U_1,\ y : A\ says\ t \vdash \\
&(\Lambda s : U_1\,.\,\lambda x : s\,.\,N)\ t\ (eval_A\ t\ y) : B\ says\ \sigma \Big\rrbracket_{\nu[\llbracket \tau \rrbracket_\nu / t][y \mapsto d]},
\end{aligned}
$$

$$
\begin{aligned}
f' = \Big\llbracket &\Gamma,\ t : U_1,\ y : A\ says\ t \vdash \\
&(\Lambda s : U_1\,.\,\lambda x : s\,.\,N)\ t\ (eval_A\ t\ y) : B\ says\ \sigma \Big\rrbracket_{\nu'[\llbracket \tau' \rrbracket_{\nu'} / t][y \mapsto d']}.
\end{aligned}
$$

From $\nu\big[[\![\tau]\!]_\nu/t\big]\big[y \mapsto d\big](y) = d$, $\nu'\big[[\![\tau']\!]_{\nu'}/t\big]\big[y \mapsto d'\big](y) = d'$, and (5),

$$\Big(\nu\big[[\![\tau]\!]_\nu/t\big]\big[y \mapsto d\big](y), \nu'\big[[\![\tau']\!]_{\nu'}/t\big]\big[y \mapsto d'\big](y)\Big) \in R_C^{[\![A \ says \ \tau]\!]_\nu, [\![A \ says \ \tau']\!]_{\nu'}}. \tag{6}$$

Moreover, according to Lemma 1, we know that

$$[\![A \ says \ \tau]\!]_\nu = [\![A \ says \ t]\!]_{\nu[[\![\tau]\!]_\nu/t][y \mapsto d]}.$$

Thus, we can rewrite (6) as

$$\Big(\nu\big[[\![\tau]\!]_\nu/t\big]\big[y \mapsto d\big](y), \nu'\big[[\![\tau']\!]_{\nu'}/t\big]\big[y \mapsto d'\big](y)\Big) \in R_C^{[\![A \ says \ t]\!]_{\nu[[\![\tau]\!]_\nu/t][y \mapsto d]}, [\![A \ says \ t]\!]_{\nu'[[\![\tau']\!]_{\nu'}/t][y \mapsto d']}}. \tag{7}$$

From Theorem 2 and (7), it is concluded that

$$(f, f') \in R_C^{[\![B \ says \ \sigma]\!]_{\nu[[\![\tau]\!]_\nu/t][y \mapsto d]}, [\![B \ says \ \sigma]\!]_{\nu'[[\![\tau']\!]_{\nu'}/t][y \mapsto d]}},$$

which is equivalent to

$$(f, f') \in R_C^{[\![B \ says \ [\tau/t]\sigma]\!]_\nu, [\![B \ says \ [\tau'/t]\sigma]\!]_{\nu'}}$$

using Lemma 1. Finally, it can be easily shown that

$$f = [\![\Gamma \vdash (\Lambda t : U_1 \,.\, \lambda x : t \,.\, N)\, \tau\, (eval_A\, \tau\, M) : B \ says \ [\tau/t]\sigma]\!]_\nu,$$

and

$$f' = [\![\Gamma \vdash (\Lambda t : U_1 \,.\, \lambda x : t \,.\, N)\, \tau'\, (eval_A\, \tau'\, M') : B \ says \ [\tau'/t]\sigma]\!]_{\nu'}.$$

- **Proof under derivation by the rule pr4**: If $P(\pi([\tau/t]\sigma,\ C))$, we have $P(\pi(B \ says \ [\tau/t]\sigma,\ C))$. As with the proof for pr3, it is derived that $A \not\preceq B$ and the type expressions $[\tau/t]\sigma$ and $[\tau'/t]\sigma$ are not protected at level A. Moreover, from $\pi(B \ says \ [\tau/t]\sigma,\ C)$, we have $\pi([\tau/t]\sigma,\ C)$. As $\pi(B \ says \ [\tau'/t]\sigma,\ C)$, at least one of the cases below holds.

  1. $C \preceq B$: The theorem has been proven for this case in the proof for pr3.

2. $\pi([\tau'/t]\sigma,\ C)$: The typing judgment $\Gamma,\ t : U_1,\ x : t \vdash N : B\ says\ \sigma$, through unitM in Figure 5, implies that there exists a term N' of type σ with free variable x of type t such that $N = \boldsymbol{\eta}_B\, N'$. Thus, from the induction hypothesis $P(\pi([\tau/t]\sigma,\ C))$, we have

$$\begin{aligned}&\Big(\llbracket\Gamma \vdash (\Lambda t : U_1 \,.\, \lambda x : t \,.\, N')\,\tau\,(eval_A\,\tau\,M) : [\tau/t]\sigma\rrbracket_\nu,\\ &\ \llbracket\Gamma \vdash (\Lambda t : U_1 \,.\, \lambda x : t \,.\, N')\,\tau'\,(eval_A\,\tau'\,M') : [\tau'/t]\sigma\rrbracket_{\nu'}\Big)\\ &\in R_C^{\llbracket[\tau/t]\sigma\rrbracket_\nu,\llbracket[\tau'/t]\sigma\rrbracket_{\nu'}}.\end{aligned} \tag{8}$$

Now, we consider the two cases $B \not\preceq C$ and $B \preceq C$. For the former,

$$\begin{aligned}R_C^{\llbracket B\ says\ [\tau/t]\sigma\rrbracket_\nu,\llbracket B\ says\ [\tau'/t]\sigma\rrbracket_{\nu'}} = {}& Dom^{\llbracket B\ says\ [\tau/t]\sigma\rrbracket_\nu} \times\\ & Dom^{\llbracket B\ says\ [\tau'/t]\sigma\rrbracket_{\nu'}},\end{aligned}$$

and thus, $P(\pi(B\ says\ \sigma,\ C))$ holds. For the latter, according to (8), we have

$$\begin{aligned}\Big(&\mathbf{T}_{B,\llbracket[\tau/t]\sigma\rrbracket_\nu}\big\llbracket\Gamma \vdash (\Lambda t : U_1 \,.\, \lambda x : A\ says\ t \,.\, N')\,\tau\,(eval_A\,\tau\,M)\\ &\qquad : [\tau/t]\sigma\big\rrbracket_\nu,\\ &\mathbf{T}_{B,\llbracket[\tau'/t]\sigma\rrbracket_\nu}\big\llbracket\Gamma \vdash (\Lambda t : U_1 \,.\, \lambda x : A\ says\ t \,.\, N')\,\tau'\,(eval_A\,\tau'\,M')\\ &\qquad : [\tau'/t]\sigma\big\rrbracket_{\nu'}\Big)\\ &\qquad \in R_C^{\llbracket B\ says\ [\tau/t]\sigma\rrbracket_\nu,\llbracket B\ says\ [\tau'/t]\sigma\rrbracket_{\nu'}}.\end{aligned}$$

Therefore, from (3),

$$\begin{aligned}\Big(&\big\llbracket\Gamma \vdash (\Lambda t : U_1 \,.\, \lambda x : A\ says\ t \,.\, N)\,\tau\,(eval_A\,\tau\,M)\\ &\qquad : B\ says\ [\tau/t]\sigma\big\rrbracket_\nu,\\ &\big\llbracket\Gamma \vdash (\Lambda t : U_1 \,.\, \lambda x : A\ says\ t \,.\, N)\,\tau'\,(eval_A\,\tau'\,M')\\ &\qquad : B\ says\ [\tau'/t]\sigma\big\rrbracket_{\nu'}\Big)\\ &\qquad \in R_C^{\llbracket B\ says\ [\tau/t]\sigma\rrbracket_\nu,\llbracket B\ says\ [\tau'/t]\sigma\rrbracket_{\nu'}}.\end{aligned}$$

□

Now, reconsider the example given in Section 2. In that example, it is assumed that there exists a proof M of $A\ says\ \sigma$ if there is a proof N of $B\ says$ READ[fileX]. It is also assumed that B is less trustworthy than A. In what follows, it is shown that deriving M from N is not possible, or equivalently, the term M is ill-typed. According to the typing rules given in Figure 5, the only way to substitute a computation for a variable in a term is the application of the rule bindM. In other words, to derive M of type $A\ says\ \sigma$ from N, N should be evaluated and substituted for some placeholder in an another term such that the result of substitution is M. This is not possible because $A\ says\ \sigma$ is not protected at level B.

6. Discussion

In a predicative polymorphic language, it is not allowed to have type variables ranging over polymorphic types. Such a language is less expressive than the one with general polymorphism in which there is no restriction on using type variables. Thus, a number of access control policies cannot be stated in a predicative language. As seen in previous sections, instead, we attain a well-defined semantics for the language, and consequently, we can define and prove a noninterference property.

Such a restriction, in particular, influences the use of *speaks for*. The expressiveness of $D^{\forall_P}$ is between simply-typed [4] and polymorphic [2] core calculi of dependency. In $D^{\forall_P}$, $A\ speaks\ for\ B$ is defined as a syntactic sugar for

$$\forall t : U_1.\ (A\ says\ t \rightarrow B\ says\ t). \tag{9}$$

The above definition for $speaks\ for$ differs from what stated in [2]. According to (9), $A\ speaks\ for\ B$ does not imply that B asserts all the statements asserted by A; only monomorphic types (statements), i.e., the ones with denotations in U_1, appear in the definition. In general, $D^{\forall_P}$ excludes the statements of the form $\forall t.\ \sigma$. As an example, it is not permitted to have

$$A\ says\ \forall t.\ (B\ says\ t \rightarrow \sigma),$$

where σ is an arbitrary predicate on t.

The abstraction of polymorphic term variables is not allowed either. This avoids having terms like $A\ speaks\ for\ B \rightarrow \sigma$ where σ is an arbitrary

Table 2: Some validities and their corresponding proof terms.

Validity	**Statement**	**Proof Term**
UNIT	$\forall t : U_1.\ t \to A\ says\ t$	$\Lambda t : U_1.\ \lambda x : t.\ \boldsymbol{\eta}_A x$
BIND	$\forall t : U_1.\ \forall s : U_1.\ (t \to A\ says\ s) \to (A\ says\ t \to A\ says\ s)$	$\Lambda t : U_1.\ \Lambda s : U_1.\ \lambda x : t \to A\ says\ s.\ \lambda y : A\ says\ t.\ x\ (\mathbf{let}\ z \Leftarrow y\ \mathbf{in}\ z)$
IDEMPOTENCE	$\forall t : U_1.\ (A\ says\ A\ says\ t \to A\ says\ t)$	$\Lambda t : U_1.\ \lambda x : A\ says\ A\ says\ t.\ \mathbf{let}\ y \Leftarrow x\ \mathbf{in}\ y$
CLOSURE	$\forall t : U_1.\ \forall s : U_1.\ (A\ says\ (t \to s)) \to ((A\ says\ t) \to (A\ says\ s))$	$\Lambda t : U_1.\ \Lambda s : U_1.\ \lambda x : A\ says\ (t \to s).\ \mathbf{let}\ x' \Leftarrow x\ \mathbf{in}\ (\lambda y : A\ says\ t.\ \mathbf{let}\ y' \Leftarrow y\ \mathbf{in}\ \boldsymbol{\eta}_A(x'y'))$
COMM	$\forall t : U_1.\ A\ says\ B\ says\ t \to B\ says\ A\ says\ t$	$\Lambda t : U_1.\ \lambda x : A\ says\ B\ says\ t.\ \mathbf{let}\ y \Leftarrow x\ \mathbf{in}\ (\mathbf{let}\ z \Leftarrow y\ \mathbf{in}\ \boldsymbol{\eta}_B(\boldsymbol{\eta}_A z))$

(a) $$\frac{N : B\ says\ \text{READ}[\mathsf{fileX}] \to A\ says\ \text{READ}[\mathsf{fileX}] \qquad \Delta : B\ says\ \text{READ}[\mathsf{fileX}]}{N\Delta : A\ says\ \text{READ}[\mathsf{fileX}]}$$

(b) $$\frac{M : A\ speaks\ for\ C \qquad \text{READ}[\mathsf{fileX}] : U_1}{M\ \text{READ}[\mathsf{fileX}] : A\ says\ \text{READ}[\mathsf{fileX}] \to C\ says\ \text{READ}[\mathsf{fileX}]}$$

(c) $$\frac{M\ \text{READ}[\mathsf{fileX}] : A\ says\ \text{READ}[\mathsf{fileX}] \to C\ says\ \text{READ}[\mathsf{fileX}] \qquad N\Delta : A\ says\ \text{READ}[\mathsf{fileX}]}{(M\ \text{READ}[\mathsf{fileX}])(N\Delta) : C\ says\ \text{READ}[\mathsf{fileX}]}$$

(d) $$\frac{P : C\ controls\ \text{READ}[\mathsf{fileX}] \qquad (M\ \text{READ}[\mathsf{fileX}])(N\Delta) : C\ says\ \text{READ}[\mathsf{fileX}]}{P((M\ \text{READ}[\mathsf{fileX}])(N\Delta)) : \text{READ}[\mathsf{fileX}]}$$

Figure 6: Example derivation of an access right.

statement. Such a limitation removes the possibility of deduction on the basis of the level of trust one may have in a principal. Thus, only part of group-based, attribute-based, and role-based access control can be implemented through our language (logic). This is due to the fact that these concepts are expressed in terms of *speaks for* [5].

More generally, there is no program of type $\tau \to \sigma$ in $D^{\forall_P}$ if there are some polymorphic types in τ or σ. As another example, consider the "hand-off axiom"

$$A\ says\ (B\ speaks\ for\ A) \to (B\ speaks\ for\ A).$$

This axiom cannot be stated in $D^{\forall_P}$ either. It is worth noting that hand-off is in doubt because of granting permissions without any control, although it has been used in several logics for access control.

Despite these restrictions, $D^{\forall_P}$ can still express many statements that may be considered valid in access control. For example, UNIT, BIND, IDEMPOTENCE, and CLOSURE [3] have proof terms in $D^{\forall_P}$. Moreover, commutativity of *says* (COMM) and the rule Necessitation also hold in the language. These validities and their corresponding proof terms are given in Table 2. Note that Necessitation corresponds to the typing rule unitM.

By an example, we show how an access right can be granted to a principal. Suppose that A, B, and C are three principals in a system in which the statements below hold; M, N, and P are assumed to be their proof terms.

$$M : A \; speaks \; for \; C$$

$$N : B \; says \; \text{READ}[\mathsf{fileX}] \rightarrow A \; says \; \text{READ}[\mathsf{fileX}]$$

$$P : C \; controls \; \text{READ}[\mathsf{fileX}]$$

From $A \; speaks \; for \; C$, we have $\forall t : U_1. A \; says \; t \rightarrow C \; says \; t$. Moreover, $C \; controls$ READ[fileX] is equivalent to $C \; says$ READ[fileX] $\rightarrow$ READ[fileX]. The statement READ[fileX] is represented by a type of universe U_1. Now, consider B requests for a read permission on fileX; the proof term for this request is Δ. Figure 6 shows how this request is granted, i.e., a proof term for READ[fileX] is derived in our language. Clearly, the conclusion of (b) is the result of type application and the ones of (a), (c), and (d) are the result of term application.

7. Conclusion

This paper proposes a predicative polymorphic calculus, and a corresponding logic, for access control in distributed systems. We also define and prove a noninterference property in this language. In fact, we provide a semantic model for our language $D^{\forall_P}$ so that it can be proven that every well-typed program of the language satisfies noninterference. The next step in our research is to extend the results of this paper to a generic calculus of dependency that supports impredicative polymorphism.

References

[1] Abadi, M.. Logic in access control. In: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science. 2003. p. 228–233.

[2] Abadi, M.. Access control in a core calculus of dependency. In: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming. 2006. p. 263–273.

[3] Abadi, M.. Variations in access control logic. In: Proceedings of the 9th International Conference on Deontic Logic in Computer Science. 2008. p. 96–109.

[4] Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.. A core calculus of dependency. In: Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages. 1999. p. 147–160.

[5] Abadi, M., Burrows, M., Lampson, B.W., Plotkin, G.D.. A calculus for access control in distributed systems. ACM Transactions on Programming Languages and Systems 1993;15(4):706–734.

[6] Abramsky, S., Jagadeesan, R.. Game semantics for access control. In: Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics. 2009. p. 135–156.

[7] Ahmed, A., Dreyer, D., Rossberg, A.. State-dependent representation independence. In: Proceedings of 36th ACM Symposium on Principles of Programming Languages (POPL). 2009. p. 340–356.

[8] Appel, A.W., Felten, E.W.. Proof-carrying authentication. In: Proceedings of the 6th ACM Conference on Computer and Communications Security. 1999. p. 52–62.

[9] Banerjee, A., Naumann, D.A.. Secure information flow and pointer confinement in a Java-like language. In: Proceedings of 15th IEEE Computer Security Foundations Workshop. 2002. p. 253–267.

[10] Banerjee, A., Naumann, D.A.. Stack-based access control and secure information flow. Journal of Functional Programming 2005;15(2):131–177.

[11] Bauer, L.. Access control for the Web via proof-carrying authorization. Ph.D. thesis; Princeton University; 2003.

[12] Birgisson, A., Russo, A., Sabelfeld, A.. Unifying facets of information integrity. In: Proceedings of the 6th International Conference on Information Systems Security. 2010. p. 48–65.

[13] Bugliesi, M., Focardi, R., Maffei, M.. Dynamic types for authentication. Journal of Computer Security 2007;15(6):563–617.

[14] Chang, R.M., Jiang, G., Ivancic, F., Sankaranarayanan, S., Shmatikov, V.. Inputs of Coma: Static detection of denial-of-service vulnerabilities. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium. 2009. p. 186–199.

[15] Fournet, C., Gordon, A., Maffeis, S.. A type discipline for authorization in distributed systems. In: Proceedings of the 20th IEEE Computer Security Foundations Symposium. 2007. p. 31–48.

[16] Fournet, C., Gordon, A.D., Maffeis, S.. A type discipline for authorization policies. In: Proceedings of the 14th European Symposium on Programming. 2005. p. 141–156.

[17] Garg, D., Pfenning, F.. Non-interference in constructive authorization logic. In: Proceedings of the 19th IEEE Computer Security Foundations Workshop. 2006. p. 283–296.

[18] Genovese, V., Garg, D.. New modalities for access control logics: Permission, control and ratification. In: Proceedings of the 7th International Workshop on Security and Trust Management. 2011. p. 56–71.

[19] Goguen, J.A., Meseguer, J.. Security policies and security models. In: Proceedings of IEEE Symposium on Security and Privacy. 1982. p. 11–20.

[20] Gordon, A.D., Jeffrey, A.. Authenticity by typing for security protocols. Journal of Computer Security 2003;11(4):451–520.

[21] Gurevich, Y., Neeman, I.. Logic of infons: The propositional case. ACM Transactions on Computational Logic 2011;12(2):9.

[22] Heintze, N., Reicke, J.G.. The SLam calculus: Programming with secrecy and integrity. In: Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages. 1998. p. 365–377.

[23] Jia, L., Vaughan, J.A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J., Zdancewic, S.. AURA: A programming language for authorization and audit. In: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming. 2008. p. 27–38.

[24] Jia, L., Zdancewic, S.. Encoding information flow in AURA. In: Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security. 2009. p. 17–29.

[25] Lafrance, S., Mullins, J.. Using admissible interference to detect denial of service vulnerabilities. In: Proceedings of the 6th International Workshop on Formal Methods. 2003. p. 1–19.

[26] Li, N., Grosof, B.N., Feigenbaum, J.. Delegation logic: A logic-based approach to distributed authorization. ACM Transactions on Information and System Security 2000;6(1):2003.

[27] Li, P., Mao, Y., Zdancewic, S.. Information integrity policies. In: Proceedings of the Workshop on Formal Aspects in Security and Trust. 2003. p. 53–70.

[28] Mitchell, J.. Foundations for programming languages. MIT Press, Cambridge, MA, USA, 1996.

[29] Mitchell, J.C.. Representation independence and data abstraction. In: Proceedings of 13th ACM Symposium on Principles of Programming Languages (POPL). 1986. p. 263–276.

[30] Moggi, E.. Notions of computation and monads. Information and Computation 1991;93(1):55–92.

[31] Pottier, F., Simonet, V.. Information flow inference for ML. In: Proceedings of 29th ACM Symposium on Principles of Programming Languages (POPL). 2002. p. 319–330.

[32] Reynolds, J.C.. Types, abstraction and parametric polymorphism. In: Mason, R.E.A., editor. Information Processing 83. Elsevier Science Publishers B. V. (North-Holland); 1983. p. 513–523.

[33] Sabelfeld, A., Myers, A.C.. Language-based information-flow security. IEEE Journal on Selected Areas in Communications 2003;21(1):5–19.

[34] Schneider, F.B., Morrisett, J.G., Harper, R.. A language-based approach to security. In: Informatics. 2001. p. 86–101.

[35] Swamy, N., Corcoran, B.J., Hicks, M.. FABLE: A language for enforcing user-defined security policies. In: Proceedings of IEEE Symposium on Security and Privacy. 2008. p. 369–383.

[36] Tse, S., Zdancewic, S.. Translating dependency into parametricity. In: Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming. 2004. p. 115–125.

[37] Vaughan, J.A.. AuraConf: A unified approach to authorization and confidentiality. In: Proceedings of 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. 2011. p. 45–58.

[38] Vaughan, J.A., Jia, L., Mazurak, K., Zdancewic, S.. Evidence-based audit. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium. 2008. p. 177–191.

[39] Volpano, D., Smith, G., Irvine, C.. A sound type system for secure flow analysis. Journal of Computer Security 1996;4(2-3):167–187.

[40] Wadler, P.. Monads for functional programming. In: Advanced Functional Programming. 1995. p. 24–52.

[41] Zdancewic, S., Myers, A.C.. Secure information flow via linear continuations. Higher Order and Symbolic Computation 2002;15(2-3):209–237.

[42] Zheng, L., Myers, A.C.. End-to-end availability policies and noninterference. In: Proceedings of 18th IEEE Computer Security Foundations Workshop. 2005. p. 272–286.

[43] Zheng, L., Myers, A.C.. Dynamic security labels and static information flow control. International Journal of Information Security 2007;6(2):67–84.