

A Logical Foundation for Authorizing Code Execution*

Maja Frydrychowicz
Supervised by Brigitte Pientka

School of Computer Science
McGill University
Montréal, Canada

Abstract

In computer security, resources can be protected by the enforcement of a formally specified access control policy. Policy enforcement implies interaction between a policy specification and program behaviour. We propose to model this interaction with type theory, and present a pure, functional programming language in which access control policies are intrinsic and statically enforced. In addition, we develop an operational semantics, and prove progress and type preservation. The underlying type system incorporates constructive authorization logic into contextual modal type theory. As a result, ordinary computation interacts with a policy expressed in authorization logic. Policy enforcement is thus integrated into the software development process, and the secure behaviour of programs is established at compile time.

1 Introduction

In computer security, access control mechanisms are designed to identify agents and assess their privileges with respect to system resources. They rely on well-crafted policy to do so, and play a key role in the protection of user privacy and the maintenance of data integrity, among other applications. On a practical scale, especially in the case of distributed systems and mobile code, the design and implementation of an access control policy is a complex, error-prone process. Many specialized authorization logics have been introduced to address this difficulty [1]. Given a policy specified in authorization logic, the availability of a permission is determined by constructing a proof of that permission from policy rules.

In a software setting, policy enforcement implies interaction between a policy specification and program behaviour. We present a functional programming language in which the formal specification of an access control policy is integrated with its implementation in a concrete setting. We achieve this by designing a type system made up of two levels: one for computation, and the other for authorization.

The computation level is based on intuitionistic contextual modal logic, as developed by Nanevski, et al [20]. Contextual modality permits reasoning about the validity of a

*September 27, 2007. Revised December 29, 2007

proposition T relative to the truth of other propositions $A \in \Psi$. Casting this idea in a computer security setting, we are able to express that a program of type T will execute only if the privileges described in Ψ are available. We describe access control policies using propositions expressed in constructive authorization logic [14]. This logic is inherently open-ended and extensible; it forms the authorization level of our type system.

The proposed language is one in which the security of a program is established at compile time. When a restricted operation is to occur, the calling program must supply a proof of authorization from the propositions that make up the local access control policy. If no such proof is provided, the program will not type-check. Security violations are therefore identified at this early stage, and can be corrected immediately. In other words, this scheme brings policy enforcement into the software development process.

Since our type system is designed in a modular fashion, its computation level may be easily integrated with more expressive extensions of constructive authorization logic. Modularity also emphasizes a clear distinction between two critical stages of secure system design: verifying the integrity of an access control policy, and verifying the integrity of a program with respect to a given policy. Furthermore, we conjecture that erasure of the authorization level as well as all security annotations at the computation level yields an ordinary pure functional language. This may imply that our approach can be built on top of an existing, well-established functional programming language.

2 Foundations

Our approach to modeling policy enforcement relies on the interaction between two distinct components. Constructive authorization logic, as presented by Garg and Pfenning [14], is used to formally specify access control policies, and to reason about the privileges they allow. We adapt intuitionistic contextual modal logic [20] to reason about the truth of a proposition relative to a context of available access privileges. Our juxtaposition of the two logics allows us typecheck an ordinary program with respect to a given access control policy.

2.1 Constructive Authorization Logic

The discussion of access control involves three basic elements: principals, resources and policy. The term *principal* generally refers to any entity capable of making statements. For example, a principal representing a web server might state a request to read a file. Such a request is granted or refused according to the policy rules in place. In authorization logic, access to a resource is granted if a proof of access is provided as evidence. Policy enforcement in our secure type system relies on this very idea.

Constructive authorization logic aims to relate truth to principal statements. Therefore, in addition to the truth of a proposition, which is independent of all principals, the logic introduces a new judgment, written $K \textit{ affirms } A$. The logic's constructive design ensures that the evidence contained in proofs of access is direct. Isolating principals as distinct entities makes it possible to model decentralized authority in distributed systems such as the World Wide Web. Each principal is able to express its own policy regarding resources under its jurisdiction. Furthermore, each principal's authority may change over time, or according to environment, or even according to other principals' statements. This model

of access control is flexible enough to allow permissions to be granted dynamically by local authorities.

We assume a scenario with known, fixed, finite sets of principals K and atomic propositions p . The form of atomic propositions is not restricted in any way. We define *affirmation* informally as the expression of a principal’s intent or policy.

Propositions: $A ::= p \mid A \supset A \mid \forall x : s. A(x) \mid K \text{ says } A \mid \perp$

Judgments: $\Sigma; \Psi \vdash A \text{ true}$
 $\Sigma; \Psi \vdash K \text{ affirms } A$

We summarize the proof-theoretic semantics of constructive authorization logic as presented in [14]. As many other logic-based models of access control, constructive authorization logic includes a *says* connective that abstracts away the details of authentication [1]. Thus, throughout our approach to policy enforcement we are able to assume that all principals are identified accurately. In constructive authorization logic, the *says* connective internalizes affirmation.

$$\frac{\Sigma; \Psi \vdash K \text{ affirms } A}{\Sigma; \Psi \vdash (K \text{ says } A) \text{ true}} \text{ says I}$$

Constructive authorization logic is also equipped with standard rules for implication and universal quantification. The signature Σ keeps track of constants and parameters such as `Tom:principal`, `c:file`, which are available for use in quantification. The domain of Σ is open-ended, but it must at least contain the kind “principal”. Similarly, Ψ represents a collection of hypotheses or assumptions A available for use in a proof.

Affirmation models principal statements directly. If A is true and K is a principal then we may conclude that $K \text{ affirms } A$.

$$\frac{\Sigma; \Psi \vdash A \text{ true} \quad \Sigma \vdash K : \text{principal}}{\Sigma; \Psi \vdash K \text{ affirms } A} \text{ AFF}$$

Next we must understand how to derive new conclusions from evidence that $K \text{ says } A$. The inference rule below states that if we have a proof that $K \text{ says } A$ is true, then in order to prove that the same principal K affirms some statement A' , we may assume that A is true to construct that proof. In other words, a statement made by principal K may only be used directly to reason about other statements made by K .

$$\frac{\Sigma; \Psi \vdash (K \text{ says } A) \text{ true} \quad \Sigma; \Psi, A \text{ true} \vdash K \text{ affirms } A'}{\Sigma; \Psi \vdash K \text{ affirms } A'} \text{ says E}$$

As an example, consider the following policy fragment expressed in constructive authorization logic:

$$\begin{aligned} &\forall x : \text{file}.(K_{\text{admin}} \text{ says CanWrite}(K_1, x)) \supset \text{CanWrite}(K_1, x) \\ &K_{\text{admin}} \text{ says CanWrite}(K_1, \text{password.txt}) \end{aligned}$$

The first rule states that principal K_{admin} is trusted with the authority to give write-access to any file it chooses to the principal K_1 . The second rule can be interpreted as K_{admin} 's own access control policy. It states that K_{admin} is giving K_1 access to write to the file *password.txt*. From these policy rules, we are able to prove that $\text{CanWrite}(K_1, \text{password.txt})$ is true.

Constructive authorization logic provides the means to analyze policies and enforce non-interference in terms of *affirmation flow*. That is, there exists a decision procedure that establishes whether the statements of a principal K can affect the conclusions of another principal K' . For instance, for the example policy fragment, one can formally verify whether additional statements made by K_1 or some other principal K' affect the conclusions made by K_{admin} regarding the permission $\text{CanWrite}(K_1, x)$, and vice versa.

Such analysis is extremely important in the design of security policies. Any policy of practical size is often too large and complex for its design to be immune to human error. Updating an existing policy to accommodate a new situation is similarly challenging. Non-interference analysis provides the means to explore the consequences of adding new rules to a policy. We endeavour to extend non-interference properties to our type-based approach to policy enforcement.

2.2 Contextual Modalities in a Security Setting

Intuitionistic contextual modal logic internalizes reasoning about validity relative to an explicitly stated context of assumptions. For instance, one may prove whether some proposition A is valid relative to a context that consists of proposition B . In other words, we can show whether A is true in any world in which B is true. This is referred to as *contextual validity* and is written as $A[B]$.

We adapt this idea to a computer security setting by instead reasoning about the type T of a program relative to the availability of some permissions Ψ' . We thus introduce the type $T[\Psi']$, which we call a *protected type*. Given a set Ψ of assumptions about security expressed in authorization logic, and a set Γ of assumptions about ordinary computation-level data, we define protected types as follows:

$$\frac{\Psi'; \Gamma \vdash T}{\Psi; \Gamma \vdash T[\Psi']} \square^{\Psi}$$

The above introduction rule states that in order to prove $T[\Psi']$, we must prove T using only the assumptions in Ψ' and Γ . Notice that Γ is not affected by this introduction rule, whereas the information in Ψ is lost, and replaced by that in Ψ' : our assumptions about data may be interpreted as true in all worlds. In our reasoning about authorization, possible worlds only differ in their security characteristics.

The next step is to define how assumptions about these protected types are to be used. The contextual hypothesis rule states that if we have assumed that T is true in any world

where Ψ' is true, then we can construct a proof of T from our current security assumptions Ψ , by proving Ψ' from Ψ :

$$\frac{\Psi \vdash \Psi'}{\Psi; \Gamma, u :: T[\Psi'] \vdash T} \text{ctxhyp}$$

To be more precise, if Ψ' consists of assumptions $\{p_1 : A_1, p_2 : A_2 \dots, p_n : A_n\}$, then we must prove each of them from Ψ :

$$\frac{\Psi \vdash A_1 \quad \Psi \vdash A_2 \quad (\dots) \quad \Psi \vdash A_n}{\Sigma; \Psi \vdash \Psi'}$$

The elimination rule for contextual types describes under what conditions we may derive new conclusions from the assumption that $T[\Psi]$ is true.

$$\frac{\Psi'; \Gamma \vdash T[\Psi] \quad \Psi'; \Gamma, u : T[\Psi] \vdash T'}{\Psi'; \Gamma \vdash T'} \square E^u$$

The above rule describes the “unlocking” of an expression that has a protected type. It states that if we can establish $T[\Psi']$ from our current assumptions Ψ, Γ , and if we can prove some proposition T' with the help of this information, then we have a proof of T' from our current assumptions.

In summary, the declaration of a protected type is described by the introduction rule, the release of a protected type is triggered by the elimination rule, and verified according to the security assumptions in Ψ by the contextual hypothesis rule.

3 A Secure Functional Programming Language

As described in the previous section, our protected types are designed to refer to permissions written in an authorization logic. Contextual modality is the means by which ordinary program data is made to interact with an access control policy. We interface constructive authorization logic with contextual modalities to produce a programming language in which well-typed expressions are guaranteed to respect the constraints of a given access control policy. This policy is specified in constructive authorization logic. Policy design and enforcement remain as separate development stages. A program will typecheck in accordance with an access control policy only if it contains the appropriate proofs regarding permission availability.

3.1 Modular Bidirectional Type System

The type system consists of two distinct, interacting levels: the computation level and the authorization level. The computation level is used to manage protected types, and to reason about ordinary data types. The authorization level is used to check proofs of access according to the inference rules of constructive authorization logic.

A program is provided with a formally specified access control policy - this may be encoded in a program specification. Type annotations are used to define the security requirements of expressions with protected type. We adapt a bidirectional type-checking algorithm in order to minimize the need for type annotations. As with the policy specification, all protected expressions may be declared via annotation in the program specification.

Typing Judgments

Our decision to use a bidirectional type checking algorithm is reflected in our typing judgments at both levels: we write $e \Leftarrow T$ to mean that the expression e checks against type T , and $e \Rightarrow T$ means that we can synthesize T from e .

Computation-level assumptions are listed in the labelled collection Γ . During the type-checking process, a program is accompanied by a previously specified access control policy with which Ψ is initialized. Protected types are managed in accordance with the policy rules as well as any other information that appears in Ψ . Types are denoted with the schematic variables T, T' , and terms with the schematic variables e, e' . The resulting typing judgments, shown below, state that the expression e has type T according to the information gathered in Ψ and Γ .

$$\Psi; \Gamma \vdash e \Rightarrow T \quad \Psi; \Gamma \vdash e \Leftarrow T$$

To form the authorization level, we assign proof terms to the natural deduction form of constructive authorization logic outlined in section 2.1. Authorization-level types represent access control rules or permissions, and are denoted with the schematic variables A, A' . Terms are denoted with the schematic variables s, s' . Here, judgments are formed according to Ψ , and the fixed set Σ which contains a list of all relevant principals and objects in the environment.

$$\Sigma; \Psi \vdash s \Rightarrow A \quad \Sigma; \Psi \vdash s \Leftarrow A$$

Although Σ does not appear in the computation-level typing judgment, we assume that protected types may only depend on permissions that refer to members of Σ . In this sense, Σ is implicit at the computation level.

Syntax and Typing

The language syntax and typing rules are summarized in Figures 1, 2, 3, and 4. In particular, the computation-level terms closely resemble those which appear in contextual modal type theory.

An expression e protected by some set of permissions Ψ' is declared with the term $\text{box } \Psi'.e$. If, in turn, we would like to unlock such a protected expression and use it in e_2 , then we write $\text{let box } u = e_1 \text{ in } e_2 \text{ end}$, where e_1 stands for a protected expression. In this sense, the elimination rule for $T[\Psi']$ corresponds to the “release” of the protected expression.

$$\frac{\Psi; \Gamma \vdash e \Leftarrow T}{\Psi'; \Gamma \vdash \text{box } \Psi.e \Leftarrow T[\Psi]} \square_{I\Psi}$$

$$\frac{\Psi'; \Gamma \vdash e \Rightarrow T[\Psi] \quad \Psi'; \Gamma, u : T[\Psi] \vdash e' \Leftarrow T'}{\Psi'; \Gamma \vdash \text{let box } u = e \text{ in } e' \text{ end} \Leftarrow T'} \square_{E^u}$$

The release, or unlocking, of a protected expression must be accompanied by a proof of access expressed using authorization-level terms. To this end, the term $u[\sigma]$ represents an

Types	$T, T' ::= \alpha \mid T \rightarrow T' \mid T[\Psi]$
Computation-Level Terms	$e, e' ::= x \mid u[\sigma] \mid \text{fn } x.e \mid e e' \mid$ $(e : T) \mid \text{rec } x.e \mid \text{box } \Psi.e \mid$ $\text{let box } u = e \text{ in } e' \text{ end}$
Simultaneous Substitution	$\sigma ::= \cdot \mid \sigma, \mathbf{s}$
Signature	$\Gamma ::= \cdot \mid \Gamma, x : T \mid \Gamma, u :: T[\Psi]$
Typing Judgments	$\Psi; \Gamma \vdash e \Rightarrow T$ $\Psi; \Gamma \vdash e \Leftarrow T$

Figure 1: Syntax and Types, Computation Level

Propositions	$A, A' ::= A \supset A' \mid K \text{ says } A \mid$ $\forall X : \tau. A(X) \mid A(w) \mid \perp$
Affirmations	$\beta ::= K \text{ affirms } A$
Security Formulas	$B ::= A \mid \beta$
Authorization-Level Terms	$\mathbf{s}, \mathbf{s}' ::= p \mid \lambda p. \mathbf{s} \mid \mathbf{s} \mathbf{s}' \mid \mathbf{s} w \mid$ $\Lambda X. \mathbf{s} \mid \text{abort} \mid (\mathbf{s} : B) \mid$ $\text{saysbind } p = \mathbf{s} \text{ in } \mathbf{s}' \text{ end}$
Signatures	$\Sigma ::= \cdot \mid \Sigma, w : \tau$ $\Psi ::= \cdot \mid \Psi, p : A$
Typing Judgments	$\Sigma; \Psi \vdash \mathbf{s} \Rightarrow B$ $\Sigma; \Psi \vdash \mathbf{s} \Leftarrow B$ $\Sigma \vdash X : \tau$

Figure 2: Syntax and Types, Authorization Level

$$\begin{array}{c}
\frac{}{\Psi; \Gamma_1, x : T, \Gamma_2 \vdash x \Rightarrow T} \text{hyp} \\
\frac{\Psi; \Gamma, x : T_1 \vdash e \Leftarrow T_2}{\Psi; \Gamma \vdash \text{fn } x.e \Leftarrow T_1 \rightarrow T_2} \rightarrow I^x \\
\frac{\Psi; \Gamma, x : T \vdash e \Leftarrow T}{\Psi; \Gamma \vdash \text{rec } x.e \Leftarrow T} \text{rec}^x \\
\frac{\Psi \vdash \sigma \Leftarrow \Psi_2}{\Psi; \Gamma, u : T[\Psi_2] \vdash u[\sigma] \Rightarrow T} \text{ctxhyp} \\
\frac{\Psi; \Gamma \vdash e \Leftarrow T}{\Psi; \Gamma \vdash (e : T) \Rightarrow T} \text{annotate} \\
\frac{\Psi; \Gamma \vdash e \Leftarrow T}{\Psi'; \Gamma \vdash \text{box } \Psi.e \Leftarrow T[\Psi]} \square I^\Psi \\
\frac{\Psi; \Gamma \vdash e \Rightarrow T' \quad T' = T}{\Psi; \Gamma \vdash e \Leftarrow T} \text{switch} \\
\frac{\Psi; \Gamma \vdash e_1 \Rightarrow T_2 \rightarrow T \quad \Psi; \Gamma \vdash e_2 \Leftarrow T_2}{\Psi; \Gamma \vdash e_1 e_2 \Rightarrow T} \rightarrow E \\
\frac{\Psi'; \Gamma \vdash e \Rightarrow T[\Psi] \quad \Psi'; \Gamma, u :: T[\Psi] \vdash e' \Leftarrow T'}{\Psi'; \Gamma \vdash \text{let box } u = e \text{ in } e' \text{ end} \Leftarrow T'} \square E^u
\end{array}$$

Figure 3: Computation Level Typing Rules

$$\begin{array}{c}
\frac{}{\Sigma; \Psi_1, p : A, \Psi_2 \vdash p \Rightarrow A} \text{a:hyp} \\
\frac{\Sigma; \Psi \vdash s_1 \Rightarrow A_2 \supset A \quad \Sigma; \Psi \vdash s_2 \Leftarrow A_2}{\Sigma; \Psi \vdash s_1 s_2 \Rightarrow A} \text{a:\supset E} \\
\frac{}{\Sigma; \Psi, p : \perp \vdash \text{abort} \Rightarrow B} \text{a:\perp E} \\
\frac{\Sigma, X : \tau; \Psi \vdash s \Leftarrow A(X)}{\Sigma; \Psi \vdash \Lambda X.s \Leftarrow \forall X : \tau. A(X)} \text{a:\forall I}^X \\
\frac{\Sigma; \Psi, p : A_1 \vdash s \Leftarrow A_2}{\Sigma; \Psi \vdash \lambda p.s \Leftarrow A_1 \supset A_2} \text{a:\supset I}^p \\
\frac{\Sigma; \Psi \vdash s \Leftarrow K \text{ affirms } A}{\Sigma; \Psi \vdash s \Leftarrow K \text{ says } A} \text{says I} \\
\frac{\Sigma; \Psi \vdash s \Leftarrow A \quad \Sigma \vdash K : \text{principal}}{\Sigma; \Psi \vdash s \Leftarrow K \text{ affirms } A} \text{a:AFF} \\
\frac{\Sigma; \Psi \vdash s \Rightarrow \forall X : \tau. A(X) \quad \Sigma \vdash w : \tau}{\Sigma; \Psi \vdash s w \Rightarrow A(w)} \text{a:\forall E} \\
\frac{\Sigma; \Psi \vdash s \Rightarrow B' \quad B' = B}{\Sigma; \Psi \vdash s \Leftarrow B} \text{a:switch} \\
\frac{\Sigma; \Psi \vdash s \Leftarrow B}{\Sigma; \Psi \vdash (s : B) \Rightarrow B} \text{A:annotation} \\
\frac{\Sigma; \Psi \vdash s \Rightarrow K \text{ says } A \quad \Sigma; \Psi, p : A \vdash s' \Leftarrow K \text{ affirms } A'}{\Sigma; \Psi \vdash \text{saysbind } p = s \text{ in } s' \text{ end} \Leftarrow K \text{ affirms } A'} \text{a: says E}^{K,p}
\end{array}$$

Figure 4: Authorization Level Typing Rules

unlocked expression with some type $T[\Psi']$, accompanied by a proof-object σ . We call u a *contextual modal variable*; its behaviour depends on modal context Ψ' . The proof-object σ serves as evidence that the release of the protected expression should indeed be granted.

$$\frac{\Psi \vdash \sigma \Leftarrow \Psi_2}{\Psi; \Gamma, u : T[\Psi_2] \vdash u[\sigma] \Rightarrow T} \text{ctxhyp}$$

We define σ as a collection of authorization-level proof terms. Each term may represent the proof of some permission A . The contextual hypothesis rule (ctxhyp) verifies whether σ proves the necessary propositions. One may think of the term $u[\sigma]$ as a *suspension* of the protected expression that it represents: a protected program is not free until its associated proof-object is checked against the governing policy contained in Ψ .

In order to express the distinction between ordinary data and suspensions of protected data, the computation-level of the type system has two kinds of variables. The definition of Γ in Figure 2 states that it may track both kinds of computation-level variables simultaneously. Variables u refer to protected expressions waiting to be unlocked, and variables x are used to represent all other data, including protected expressions that have not yet been highlighted for release.

In addition to standard definitions for λ -abstraction and recursion, the computation level has terms and typing rules that make bidirectional type-checking possible. Annotation of the term e with type T is written as $(e : T)$. Analogous term annotation appears at the authorization level. The *switch* rule characterizes non-normal programs.

4 Example

Running Protected Code

Recall the policy presented in Section 2.1 and consider a scenario in which a function named `deletePasswords` requires the permission `CanWrite(K_1 , password.txt)` to be present whenever this function is used. In order apply `deletePasswords` to the file *password.txt*, we write the following code:

```
let box
  u = deletePasswords
in
  u[proof] "password.txt"
end
```

The authorization context Ψ is initialized with the rules of the governing policy:

$$p_1 : \forall x : \text{file}. (K_{\text{admin}} \text{ says } \text{CanWrite}(K_1, x)) \supset \text{CanWrite}(K_1, x)$$

$$p_2 : K_{\text{admin}} \text{ says } \text{CanWrite}(K_1, \textit{password.txt})$$

Let $A = \text{CanWrite}(K_1, \textit{password.txt})$, $\text{dP} = \text{deletePasswords}$ and $\text{p.txt} = \textit{password.txt}$. We assume that Σ contains the entry *password.txt*:file. In this environment, the above code typechecks as follows:

$$\frac{\frac{\Psi; \Gamma \vdash \text{dP} \Rightarrow (T_1 \rightarrow T_2)[A] \quad \Psi; \Gamma, u :: (T_1 \rightarrow T_2)[A] \vdash (u[\text{proof}] \text{"p.txt"}) \Leftarrow T_2}{\Psi; \Gamma \vdash \text{let box } u = \text{dP in } (u[\text{proof}] \text{"p.txt"}) \text{ end} \Leftarrow T_2} \text{hyp}}{\Psi; \Gamma \vdash \text{let box } u = \text{dP in } (u[\text{proof}] \text{"p.txt"}) \text{ end} \Leftarrow T_2} \square E^u$$

where $\text{proof} = (p_1 (\text{p.txt})) (p_2)$ is the proof object that serves as evidence for proposition A . The derivation \mathcal{D} , which completes the above typing derivation, demonstrates how the proof of access is ultimately triggered by the `letbox` construct.

$$\mathcal{D} = \frac{\frac{\frac{\vdots}{\Sigma; \Psi \vdash \text{proof} \Leftarrow A} \quad \Psi; \Gamma, u :: (T_1 \rightarrow T_2)[A] \vdash u[\text{proof}] \Leftarrow T_1 \rightarrow T_2 \quad \frac{\Psi; \Gamma, u :: (T_1 \rightarrow T_2)[A] \vdash \text{"p.txt"} \Rightarrow T_1}{\Psi; \Gamma, u :: (T_1 \rightarrow T_2)[A] \vdash \text{"p.txt"} \Leftarrow T_1} \text{hyp}}{\Psi; \Gamma, u :: (T_1 \rightarrow T_2)[A] \vdash u[\text{proof}] \Leftarrow T_1 \rightarrow T_2} \text{ctxhyp}}{\Psi; \Gamma, u :: (T_1 \rightarrow T_2)[A] \vdash (u[\text{proof}] \text{"p.txt"}) \Leftarrow T_2} \text{switch} \rightarrow E$$

The term supplied in `proof` does indeed make valid evidence for $A = \text{CanWrite}(K_1, \text{password.txt})$, and it is constructed from the rules in Ψ . Therefore, the whole expression typecheck with type T_2 . However, in this same environment, if we try to apply another function `deleteLog` that requires some other permission $\text{CanWrite}(K_1, \text{logfile.txt})$, type checking will fail: even if data of the right type is provided to `deleteLog`, there is no way to construct an authorization-level proof that $\text{CanWrite}(K_1, \text{logfile.txt})$ is available under the current policy.

5 Operational Semantics

To a great extent, the transitional semantics for our programming language match standard call-by-name semantics. The rules are summarized in Figure 5. However, the most important transition rules, namely those which describe the evaluation of unlocked protected expressions, require the definition of non-standard free variables and corresponding substitutions. In this section, we review the most important definitions and discuss progress and type preservation. The remaining details may be found in the appendix.

Overall, there are four substitution definitions: one for each kind of variable in our system. Substitution for ordinary variables x is used in λ -abstraction and recursion at the computation level. Substitution for contextual modal variables u characterizes the evaluation of protected expressions. Substitution for authorization-level variables p is applied to proof objects, as well as λ -abstraction and binding at the authorization level.

Protected expressions must be treated with special care. We thus focus the present account on contextual modal variables, and the interaction between protected expressions and ordinary, as well as authorization-level, variables.

Let us first consider the case of substitution on a protected expression. By virtue of being protected by some set of permissions Ψ' , all of its authorization-level variables are bound by Ψ' . Therefore, applying substitution for an authorization-level variable has no effect:

$$[s/p](\text{box } \Psi'.e) = \text{box } \Psi'.e$$

On the other hand, substitution for an ordinary variable x , or for a contextual modal variable u , proceed as usual:

$$[e'/x](\text{box } \Psi'.e) = \text{box } \Psi'.[e'/x](e)$$

$$[\Psi'.e'/u](\text{box } \Psi'.e) = \text{box } \Psi'.[\Psi'.e'/u](e)$$

Next we turn to general substitution for contextual modal variables. The evaluation rule that corresponds to the release of a protected expression represented by the contextual modal variable u is as follows:

$$\text{let box } u = \text{box } \Psi'.e \text{ in } e' \text{ end} \mapsto [\Psi'.e/u]e'$$

The rule states that we substitute the protected expression e for u in e' *in the context* Ψ' . This distinction about context is crucial. The substitution will proceed inside e' until it reaches the term $u[\sigma]$, which is a suspension of our protected expression accompanied by σ , a proof of Ψ' . The protected expression may, in turn, contain other proof objects σ' that may refer to variables in Ψ' . To account for this possibility, we must substitute the appropriate proof terms in σ for any references to variables in Ψ' . This is expressed as follows:

$$[\Psi'.e/u](u[\sigma]) = [\sigma|\Psi]e \quad \text{where } \sigma|\Psi \text{ represents substitution domain recovery}$$

Domain recovery refers to endowing σ with a substitution domain: in the case of $\sigma|\Psi$, each term in σ is assigned to be substituted for a variable in Ψ . The notation $[\sigma|\Psi]$ refers to a simultaneous substitution for authorization-level variables, which substitutes terms in σ for all occurrences of variables from Ψ' . If the target expression does not contain any proof objects, or any variables from Ψ' , the proof object σ is simply erased, and we are left with the successfully released expression e .

Based on these definitions, we have been able to show progress and type preservation for the operational semantics. For the sake of brevity, we write \circ to mean any of \Rightarrow or \Leftarrow .

Theorem 1 (Progress)

1. If $e \circ T$ then either e is a value or $\exists e'$ such that $e \mapsto e'$.

Proof. In each case, we proceed by induction on the derivation of $s \circ B$ and $e \circ T$ respectively.

Theorem 2 (Type Preservation)

1. If $s \circ B$ and $s \mapsto s'$ then $s' \circ B$.
2. If $e \circ T$ and $e \mapsto e'$ then $e' \circ T$.

Proof. In each case, we proceed by induction on the derivation of $s \circ B$ and $e \circ T$ respectively. Note that the induction for 2.2 depends on result 2.1.

6 Related Work

Constructive authorization logic [14] and contextual modal type theory [20] form the foundations of this work. The relationship between modal logic and security, at the level of information leaks in program data, has been studied by Miyamoto and Igarashi [19]. They describe legal directions of information flow as a reachability relation among possible worlds in which every world corresponds to a security level. Their interpretation of modal necessity is similar to our interpretation of contextual modal necessity: necessity introduction and elimination are compared to “sealing” and “unsealing” of protected information. In contrast with our work, their typing judgments are indexed by the elements of a fixed lattice of security levels, whereas we are able to reason about types relative to a flexible, custom-built policy.

The Java Security Architecture implements an access control mechanism that is based on an algorithm called *stack inspection* [15]. The algorithm makes access control decisions relative to a collection of protection domains by examining the run-time stack for the appropriate security annotations. It is triggered by special function calls placed right before a restricted method is to be called. This approach has proved reasonably effective in practice, although it is not clear what is guaranteed in terms of security because the process takes place at run time [27]. However, the algorithm appears to address two classic security problems successfully. For this reason, we believe it would be useful to formulate a practical comparison between the stack inspection model and our approach some time in the future. There exist several proposals for static, type-based encodings of stack inspection for imperative programming. Pottier, Skalka and Smith model a simplified version of the Java Security Architecture with several constraint and unification-based type systems [22]. Their types are annotated with evaluation contexts that implicitly represent the run-time stack. Fournet and Gordon present a somewhat related formal semantics for stack inspection, this time based on reasoning about intersections of sets of permissions [11].

The concept of a general security language was introduced by de TREVILLE in 2002 [9]. Such a language can potentially be used to express security statements in a distributed system, but it does not relate policy to computation. The SLam calculus is a typed λ -calculus that maintains security information as well as type information [17], and its design shares many goals with our approach. The type system propagates security information to capture both access control and information flow in a functional programming language with side-effects and concurrency. However, the prevention of security violations relies heavily on typing annotations.

Our approach bears resemblance to proof-carrying code [21] in that we associate each program with a proof of access to protected types, as needed. Proof-carrying code, as described by Necula in 1997, refers to a method of binding a program binary with a formal proof of having satisfied general specification requirements. The safety policy is formalized using an extended first-order predicate logic. A proof is joined to a program at compile time by the “code producer”, and checked at run time by the “code consumer”. The incorporation of access proofs into our type system moves the proof checking stage from run time to compile time.

7 Extensions & Future Work

Non-Interference

As mentioned in Section 2.1, it is possible to analyze a policy specified in constructive authorization logic for the presence of affirmation flow non-interference properties. Such analysis is useful in the construction of large, complex policies, which are inherently vulnerable to design flaws. We have built our type system with non-interference analysis in mind, since a policy enforcement mechanism is only effective if it maintains all the properties of the specified policy. Given the localized manner in which computation-level types interact with the authorization level, we predict that a proof of non-interference property invariance for our current type system will be a reasonably straightforward induction on derivations. However, as we expand the language with more functionality, extending such a result will be significantly more challenging, and will guide how we proceed with our design.

Code Ownership

Policy enforcement in a software setting requires that we define a clear relationship between a policy specification and a body of code. Constructive authorization logic uses principals to model decentralized authority. Currently, we only describe how propositions in authorization logic relate to parts of a program via the notion of protected types at the computation level. An expression e that requires permission A to be available is associated with that permission. In a sense, A represents the expression e at the authorization level and in the policy specification. However, so far, principals only play an informal role in our type system, which limits our ability to model access control in distributed systems. One might explore the consequences of reasoning about code ownership with contextual modalities by incorporating principals into our protected types. Formalizing code ownership will make our model of the relationship between policy and computation more accurate. For instance, we intend to address the classic Confused Deputy Problem [16], which describes a scenario in which trusted code mistakenly performs a protected operation on behalf of an untrusted piece of code.

Dynamic Policy Updates

A fixed policy is impractical for any long-running program such as a web server or an operating system. Enabling such programs to trigger changes in the governing policy is highly desirable. Furthermore, in any program, it is best to enable the availability of privileges only when they are needed for a specific task. This is called the Principle of Least Privilege [23].

It is possible to model the addition of new permissions via delegation by a trusted principal. Constructive authorization logic is able to express delegation with the help of principal statements and universal quantification. For example, if K_1 has authority over the entire file system, we can implement a policy rule that states that K_1 may delegate its access rights to file x to any university employee of its choice. A university employee then acquires these new access rights only when and if K_1 says so. In this manner, dynamic policy updates are triggered by principal statements in a controlled manner. Therefore, we may be able to achieve this functionality in our language by encoding principal statements at the computation level of the type system.

Erasure-Based Translation

The modular design of our programming language indicates that it may be possible to convert a program written in a pure functional programming language into a form that type checks according to our system and vice versa. Investigating the consequences of erasing the authorization level and all policy-related typing annotations may lead to an appropriate translation. If this is the case, we will be able to build our authorization mechanisms on top of an existing functional language.

8 Conclusion

We have presented a pure functional language in which well-typed programs are guaranteed to adhere to the rules of a given access control policy. Our type system and operational semantics exhibit progress and type preservation. Security violations may be identified and corrected at compile time, and thus policy enforcement is made a part of the program development process. By adapting contextual modal type theory to interact with a constructive authorization logic, we have built a two-level type system that emphasizes the distinction between policy design and enforcement. This modular design is also easily extensible with a more expressive authorization logic. We have developed a theoretical foundation for reasoning about the type of an expression relative to a context of access control permissions. Our definition of protected types models the interaction between a formal policy specification and the computational behaviour of programs.

References

- [1] Martín Abadi. Logic in access control. In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 228, Washington, DC, USA, 2003. IEEE Computer Society.
- [2] Martín Abadi. Access control in a core calculus of dependency. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 263–273, New York, NY, USA, 2006. ACM Press.
- [3] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, 1993.
- [4] Anindya Banerjee and David A. Naumann. A simple semantics and static analysis for java security. CS Report 2001-1, July 2001.
- [5] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, pages 93–108, Berkeley, CA, USA, 2002. USENIX Association.
- [6] Andrew Cirillo, Radha Jagadeesan, Corin Pitcher, and James Riely. Do as i say! programmatic access control with explicit identities. In *In proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 16–30, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
- [7] Jason Crampton, George Loizou, and Greg O’Shea. A logic of access control. *The Computer Journal*, 44(1):54–66, 2001.
- [8] Karl Crary, Aleksey Kligler, and Frank Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15(2):249–291, 2005.
- [9] John DeTreville. Binder, a logic-based security language. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2002. IEEE Computer Society.
- [10] Cedric Fournet, Andrew Gordon, and Sergio Maffeis. A type discipline for authorization in distributed systems. In *In proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 31–48, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

- [11] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst.*, 25(3):360–399, 2003.
- [12] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In M. Sagiv, editor, *Programming Languages and Systems*, volume 3444/2005 of *Lecture Notes in Computer Science*, pages 141–156. Springer Verlag, Berlin Heidelberg, 2005.
- [13] Deepak Garg, Lujo Bauer, Kevin D. Bowers, Frank Pfenning, and Michael K. Reiter. A linear logic of authorization and knowledge. In *Computer Security ESORICS 2006*, volume 4189/2006 of *Lecture Notes in Computer Science*, pages 297–312. Springer Verlag, Berlin Heidelberg, 2006.
- [14] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 283–296, Washington, DC, USA, 2006. IEEE Computer Society.
- [15] L. Gong and R. Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Internet Society Symposium on Network and Distributed System Security*, pages 125–134, San Diego, CA, 1998.
- [16] Normal Hardy. The confused deputy. *ACM Oper. Syst. Rev.*, 22(4):36–38, 1988.
- [17] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998*, pages 365–377, New York, NY, USA, 1998. ACM Press.
- [18] Butler W. Lampson. Computer security in the real world. *Computer*, 37(6):37–46, 2004.
- [19] Kenji Miyamoto and Atsushi Igarashi. A modal foundation for secure information flow. In *In Proceedings of the Workshop on Foundations of Computer Security (FCS'04)*, pages 187–203, July 2004.
- [20] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. To appear, 2007.
- [21] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.
- [22] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. *ACM Trans. Program. Lang. Syst.*, 27(2):344–382, 2005.
- [23] M.D. Saltzer, J.H.; Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*.
- [24] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 202–216, Washington, DC, USA, 2006. IEEE Computer Society.
- [25] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 179–193, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
- [26] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.
- [27] Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol.*, 9(4):341–378, 2000.
- [28] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for java. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 116–128, New York, NY, USA, 1997. ACM Press.
- [29] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *SIGOPS Oper. Syst. Rev.*, 27(5):256–269, 1993.

Appendix I

Substitutions

The type system makes use of several different kinds of variables. “Ordinary variables” x are used in λ -abstraction and recursion at the computation level. Contextual modal variables u refer to restricted expressions at the computation level. Authorization-level variables p refer to terms at the authorization-level. Sorting variables X are used as part of universal quantification. We must define distinct substitutions for each of these variables.

$[e/x]$	substitution for ordinary variables
$[\Psi.e/u]$	substitution for contextual modal variables
$[s/p]$	substitution for authorization-level variables
$[w/X]$	substitution for sorting variables

Definition (domain recovery $\sigma|\Psi$)

Given $\sigma = (s_1, \dots, s_n)$ and $\Psi = (p_1 : A_1, \dots, p_n : A_n)$

$$\begin{aligned} \cdot|\Psi &= \cdot && \forall\Psi \\ \sigma|\Psi &= (s_1/p_1, \dots, s_n/p_n) \end{aligned}$$

$u[\sigma]$ is a postponed substitution; it does not have a domain assigned to it yet.

Definition (simultaneous substitution $[\sigma|\Psi]e$)

$$\begin{aligned}
[\sigma_1|\Psi, \mathbf{s}/p, \sigma_2|\Psi](p) &= \mathbf{s} \\
[\sigma|\Psi](w) &= w \\
[\sigma|\Psi](X) &= X \\
[\sigma|\Psi](\lambda p.\mathbf{s}) &= \lambda p.([\sigma|\Psi, p/p]\mathbf{s}) \text{ if } p \notin \text{dom}(\sigma|\Psi) \\
[\sigma|\Psi](\mathbf{s}_1 \mathbf{s}_2) &= ([\sigma|\Psi]\mathbf{s}_1) ([\sigma|\Psi]\mathbf{s}_2) \\
[\sigma|\Psi](\text{saysbind } q = \mathbf{s}_1 \text{ in } \mathbf{s}_2 \text{ end}) &= \text{saysbind } q = [\sigma|\Psi]\mathbf{s}_1 \text{ in } [\sigma|\Psi, q/q]\mathbf{s}_2 \text{ end} \\
&\quad \text{if } q \notin \text{dom}(\sigma|\Psi), q \notin \text{FAV}(\sigma) \\
[\sigma|\Psi](\cdot) &= (\cdot) \\
[\sigma|\Psi](\sigma', \mathbf{s}) &= ([\sigma|\Psi]\sigma', [\sigma|\Psi]\mathbf{s}) \\
[\sigma|\Psi](\Lambda X.\mathbf{s}) &= \Lambda X.([\sigma|\Psi]\mathbf{s}) \\
[\sigma|\Psi](\text{abort}) &= \text{abort} \\
[\sigma|\Psi](\langle \mathbf{s} : B \rangle) &= ([\sigma|\Psi](f\mathbf{s}) : B) \\
[\sigma|\Psi](\mathbf{s} w) &= ([\sigma|\Psi](\mathbf{s}) w) \\
\\
[\sigma|\Psi](x) &= x \\
[\sigma|\Psi](\text{fn } y.\mathbf{e}) &= \text{fn } y.([\sigma|\Psi]\mathbf{e}) \\
[\sigma|\Psi](\mathbf{e}_1 \mathbf{e}_2) &= ([\sigma|\Psi]\mathbf{e}_1) ([\sigma|\Psi]\mathbf{e}_2) \\
[\sigma|\Psi](u[\sigma']) &= u[[\sigma|\Psi]\sigma'] \\
[\sigma|\Psi](\text{rec } x.\mathbf{e}) &= \text{rec } x.([\sigma|\Psi]\mathbf{e}) \\
[\sigma|\Psi](\text{box } \Psi'.\mathbf{e}) &= \text{box } \Psi'.\mathbf{e} \\
[\sigma|\Psi](\langle \mathbf{e} : T \rangle) &= ([\sigma|\Psi]\mathbf{e} : T) \\
[\sigma|\Psi](\text{let box } u = \mathbf{e}_1 \text{ in } \mathbf{e}_2 \text{ end}) &= \text{let box } x = ([\sigma|\Psi]\mathbf{e}_1) \text{ in } ([\sigma|\Psi]\mathbf{e}_2) \text{ end}
\end{aligned}$$

Definition (Substitution for ordinary variables)

$$\begin{aligned}
[e/x](x) &= e \\
[e/x](y) &= y \quad \text{if } x \neq y \\
[e/x](\text{fn } y.e') &= \text{fn } y.([e/x](e')) \quad \text{if } x \neq y \text{ and } y \notin \text{FOV}(e) \\
[e/x](e_1 e_2) &= ([e/x](e_1) [e/x](e_2)) \\
[e/x](\text{rec } y.e') &= \text{rec } y.([e/x](e')) \quad \text{if } x \neq y \text{ and } y \notin \text{FOV}(e) \\
[e/x](\text{box } \Psi.e') &= \text{box } \Psi.([e/x](e')) \\
[e/x](\text{let box } u = e_1 \text{ in } e_2 \text{ end}) &= \text{let box } u = [e/x](e_1) \text{ in } [e/x](e_2) \text{ end, if } u \notin \text{FMV}(e) \\
[e/x](u[\sigma]) &= u[\sigma] \\
[e/x](s) &= s \quad \forall \text{ security statements } s
\end{aligned}$$

Definition (Substitution for contextual modal variables)

$$\begin{aligned}
[\Psi.e/u](x) &= x \\
[\Psi.e/u](\text{fn } y.e') &= \text{fn } y.([\Psi.e/u](e')) \\
[\Psi.e/u](e_1 e_2) &= ([\Psi.e/u](e_1) [\Psi.e/u](e_2)) \\
[\Psi.e/u](\text{rec } x.e') &= \text{rec } x.([\Psi.e/u](e')) \\
[\Psi.e/u](\text{box } \Psi'.e') &= \text{box } \Psi'.([\Psi.e/u](e')) \\
[\Psi.e/u](\text{let box } v = e_1 \text{ in } e_2 \text{ end}) &= \text{let box } v = [\Psi.e/u](e_1) \text{ in } [\Psi.e/u](e_2) \text{ end} \\
&\quad \text{if } u \neq v \text{ and } v \notin \text{FMV}(e) \\
[\Psi.e/u](v[\sigma]) &= v[\sigma] \text{ if } u \neq v \\
[\Psi.e/u](u[\sigma]) &= [\sigma|\Psi]e \quad \text{where } \sigma|\Psi \text{ represents substitution} \\
&\quad \text{domain recovery.} \\
[\Psi.e/u](s) &= s \quad \forall \text{ security statements } s \\
[\Psi.e/u](\sigma, s') &= (\sigma, s') \\
[\Psi.e/u](\cdot) &= (\cdot)
\end{aligned}$$

Definition (Substitution for authorization-level variables.)

$[s/p](x)$	$= x$
$[s/p](\text{fn } y.e')$	$= \text{fn } y.[s/p](e')$
$[s/p](e_1 e_2)$	$= ([s/p](e_1) [s/p](e_2))$
$[s/p](\text{rec } x.e')$	$= \text{rec } x.[s/p](e')$
$[s/p](\text{box } \Psi.e)$	$= \text{box } \Psi.e$
$[s/p](\text{let box } u = e_1 \text{ in } e_2 \text{ end})$	$= \text{let box } u = [s/p](e_1) \text{ in } [s/p](e_2) \text{ end}$
$[s/p](e : T)$	$= ([s/p](e) : T)$
$[s/p](u[\sigma])$	$= u[[s/p](\sigma)]$
$[s/p](\sigma, s')$	$= ([s/p](\sigma), [s/p](s'))$
$[s/p](\cdot)$	$= (\cdot)$
$[s/p](p)$	$= s$
$[s/p](q)$	$= q \text{ if } p \neq q$
$[s/p](w)$	$= w$
$[s/p](X)$	$= X$
$[s/p](\lambda q.s')$	$= \lambda q.[s/p](s') \text{ if } p \neq q \text{ and } q \notin \text{FAV}(s)$
$[s/p](s_1 s_2)$	$= ([s/p](s_1) [s/p](s_2))$
$[s/p](\text{saysbind } q = s_1 \text{ in } s_2 \text{ end})$	$= \text{saysbind } q = [s/p](s_1) \text{ in } [s/p](s_2) \text{ end}$ $\text{if } p \neq q \text{ and } q \notin \text{FAV}(s)$
$[s/p](\Lambda X.s')$	$= \Lambda X.([s/p](s'))$
$[s/p](\text{abort})$	$= \text{abort}$
$[s/p](s' : B)$	$= ([s/p](s') : B)$
$[s/p](s' w)$	$= ([s/p](s') w)$

Definition (Substitution for sorting variables.)

$$\begin{aligned}
[w/X](p) &= p \\
[w/X](w') &= w' \\
[w/X](X) &= w \\
[w/X](X') &= X' \text{ if } X \neq X', w \neq X', X' \text{ not free in } w \\
[w/X](\lambda q.s') &= \lambda q.[w/X](s') \\
[w/X](s_1 s_2) &= ([w/X](s_1) [w/X](s_2)) \\
[w/X](\text{saysbind } q = s_1 \text{ in } s_2 \text{ end}) &= \text{saysbind } q = [w/X](s_1) \text{ in } [w/X](s_2) \text{ end} \\
[w/X](\Lambda X'.s') &= \Lambda X'.([w/X](s')) \quad \text{if } X \neq X' \\
[w/X](\text{abort}) &= \text{abort} \\
[w/X](s' : B) &= ([w/X](s') : B) \\
[w/X](s' w') &= ([w/X](s') w') \\
\\
[w/X](\forall X' : \tau.A(X')) &= \forall Y : \tau.[w/X](A(Y)) \quad X \neq X' \\
[w/X](\perp) &= \perp \\
[w/X](A \supset A') &= [w/X](A) \supset [w/X](A') \\
[w/X](Y \text{ says } A) &= [w/X](Y) \text{ says } [w/X](A)
\end{aligned}$$

Free Variables

The above substitutions refer to three different kinds of free variables: free ordinary variables (FOV), free contextualmodal variables (FMV) and free authorization variables (FAV). These free variables are defined here.

$\text{FOV}(\text{fn } x.e)$	$= \text{FOV}(e) \setminus \{x\}$
$\text{FOV}(e_1 e_2)$	$= \text{FOV}(e_1) \cup \text{FOV}(e_2)$
$\text{FOV}(\text{rec } x.e)$	$= \text{FOV}(e) \setminus \{x\}$
$\text{FOV}(\text{box } \Psi.e)$	$= \text{FOV}(e)$
$\text{FOV}(\text{let box } u = e \text{ in } e' \text{ end})$	$= \text{FOV}(e) \cup \text{FOV}(e')$
$\text{FOV}((e: T))$	$= \text{FOV}(e)$
$\text{FOV}(u[\sigma])$	$= \{ \}$
$\text{FOV}(s)$	$= \{ \}$ for any security statement s
$\text{FOV}(\cdot)$	$= \{ \}$
$\text{FOV}(\sigma, s)$	$= \{ \}$

$\text{FMV}(\text{fn } x.e)$	$= \text{FMV}(e)$
$\text{FMV}(e_1 e_2)$	$= \text{FMV}(e_1) \cup \text{FMV}(e_2)$
$\text{FMV}(\text{rec } x.e)$	$= \text{FMV}(e)$
$\text{FMV}(\text{box } \Psi.e)$	$= \text{FMV}(e)$
$\text{FMV}(\text{let box } u = e \text{ in } e' \text{ end})$	$= \text{FMV}(e) \cup \text{FMV}(e') \setminus \{u\}$
$\text{FMV}((e: T))$	$= \text{FMV}(e)$
$\text{FMV}(u[\sigma])$	$= \{u\}$
$\text{FMV}(s)$	$= \{ \}$ for any authorization-level term s
$\text{FMV}(\cdot)$	$= \{ \}$
$\text{FMV}(\sigma, s)$	$= \{ \}$

$\text{FAV}(x)$	$= \{\}$
$\text{FAV}(\text{fn } x.e)$	$= \text{FAV}(e)$
$\text{FAV}(e_1 e_2)$	$= \text{FAV}(e_1) \cup \text{FAV}(e_2)$
$\text{FAV}(\text{rec } x.e)$	$= \text{FAV}(e)$
$\text{FAV}(\text{box } \Psi.e)$	$= \{\}$
$\text{FAV}(\text{let box } u = e \text{ in } e' \text{ end})$	$= \text{FAV}(e) \cup \text{FAV}(e')$
$\text{FAV}((e: T))$	$= \text{FAV}(e)$
$\text{FAV}(u[\sigma])$	$= \text{FAV}(\sigma)$
$\text{FAV}(p)$	$= \{p\}$
$\text{FAV}((s: A))$	$= \text{FAV}(s)$
$\text{FAV}(\lambda p.s)$	$= \text{FAV}(s) \setminus \{p\}$
$\text{FAV}(s_1 s_2)$	$= \text{FAV}(s_1) \cup \text{FAV}(s_2)$
$\text{FAV}(\text{saysbind } p = s \text{ in } s' \text{ end})$	$= \text{FAV}(s) \cup \text{FAV}(s') \setminus \{p\}$
$\text{FAV}(\cdot)$	$= \{\}$
$\text{FAV}(\sigma, s)$	$= \text{FAV}(\sigma) \cup \text{FAV}(s)$

$$\begin{array}{c}
\text{Values } v ::= \text{fn } x.v \mid \text{box } \Psi.e \\
\\
(\text{fn } x.e) (e') \mapsto [e'/x]e \qquad \frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \\
\\
\text{rec } x.e \mapsto [\text{rec } x.e/x]e \qquad \frac{e \mapsto e'}{(e : T) \mapsto (e' : T)} \\
\\
\text{let box } u = \text{box } \Psi.e \text{ in } e' \text{ end} \mapsto [\Psi.e/u]e' \\
\\
\frac{e_1 \mapsto e'_1}{\text{let box } u = e_1 \text{ in } e_2 \text{ end} \mapsto \text{let box } u = e'_1 \text{ in } e_2 \text{ end}}
\end{array}$$

Figure 5: Transition Semantics, Computation Level