Evidence-based Audit

Jeffrey A. Vaughan Limin Jia Karl Mazurak Steve Zdancewic

University of Pennsylvania

Abstract

Authorization logics provide a principled and flexible approach to specifying access control policies. One of their compelling benefits is that a proof in the logic is evidence *that an access-control decision has been made in accordance with policy. Using such proofs for auditing purposes is implicit in much of the work on authorization logics and proof-carrying authorization.*

This paper explores some ramifications of adopting this "proofs as log entries" approach to auditing. Two benefits of evidence-based audit are a reduced trusted computing base and the ability to detect flaws in complex authorization policies. Moreover, the proof structure is itself useful, because operations like proof normalization can yield information about the relevance of policy statements.

To explain these observations concretely, we develop a rich authorization logic based on a dependently-typed variant of DCC and prove the metatheoretic properties of subject-reduction and normalization. We show untrusted but well-typed applications, that access resources through an appropriate interface, must obey the access control policy and create proofs useful for audit. We show the utility of proof-based auditing in a number of examples and discuss several pragmatic issues, such as proof size, that must be addressed in this context.

1 Introduction

Auditing, *i.e.* recording significant events that occur during a system's execution for subsequent review, has long been recognized as a crucial part of building secure systems. A typical use of auditing is found in a firewall, which might keep a log of the access control decisions that it makes when deciding whether to permit connection requests. In this case, the log might consist of a sequence of time stamped strings written to a file where each entry indicates some information about the nature of the request (IP addresses, port numbers, *etc.*) and whether the request was permitted. Other scenarios place more stringent requirements on the log. For example, a bank server's transactions log should be tamper resistant, and log entries should be authenticated and not easily forgeable. Audit logs are useful because they can help administrators (or other auditors) to both identify sources of unusual or malicious behavior and to find flaws in the authorization policies enforced by the system.

Despite the importance of auditing in practice, there has been surprisingly little research into what constitutes good auditing procedures.[1] There has been work on cryptographically protecting logs to prevent or detect log tampering [32, 15], efficiently searching confidential logs [33], and experimental research on effective, practical logging [9, 29]. But there is relatively little work on *what* the contents of an audit log should be or how to ensure that a system implementation performs appropriate logging (see Wee's paper on a logging and auditing file system [34] for one approach to these issues, however).

In this paper, we argue that audit log entries should constitute *evidence* that justifies the authorization decisions made during the system's execution. What is valid evidence for an authorization decision? Following an abundance of prior work on authorization logic [3, 28, 21, 1, 30, 2, 25], we adopt the stance that log entries should contain *proofs* of suitable propositions that encode access-control queries. Indeed, the idea of logging such proofs is implicit in the proof-carrying authorization literature [7, 11, 14], but, to our knowledge, the use of proofs for auditing purposes has not been studied outright.

There are several compelling reasons why it is profitable to include proofs of authorization decisions in the log. First, by connecting the contents of log entries directly to the authorization policy (as expressed by a collection of rules stated in terms of the authorization logic), we obtain a principled way of determining what information to log. Second, proofs contain structure that can potentially help administrators find flaws or misconfigurations in the authorization policy. Third, storing verifiable evidence helps reduce the

[1] Note that the term auditing can also refer to the practice of *statically* validating a property of the system. Code review, for example, seeks to find flaws in software before it is deployed. Such auditing is, of course, very important, but this paper focuses on *dynamic* auditing mechanisms such as logging.

size of the trusted computing base, since the justification for the authorization decision made is available for independent inspection. We shall consider all of these benefits in more detail in Section 2.

The impetus for this paper stems from our experience with the (ongoing) design and implementation of a new security-oriented programming language called Aura. Among other features intended to make building secure software easier, Aura provides a built-in notion of principals, and its type system includes authorization proofs as first-class objects; the authorization policies may themselves depend on program values. The Aura runtime system will use authorization proofs when constructing audit logs.

One goal of this paper is to investigate some of the ramifications of adopting the design decision to log proof objects. A second goal is to find mechanisms that can be used both to simplify the task of manipulating authorization proofs and to ensure that (potentially untrusted) software performs appropriate logging. This paper therefore focuses on the use of proofs for logging purposes and the way in which we envision structuring Aura software to take advantage of the authorization policies to minimize the trusted computing base. The main contributions of this paper can be summarized as follows.

Section 2, proposes a system architecture in which logging operations are performed by a trusted kernel, which can be thought of as part of the Aura runtime system. Such a kernel accepts proof objects constructed by programs written in Aura and logs them while performing security-relevant operations.

To illustrate Aura more concretely, Section 3 develops a dependently typed authorization logic based on DCC [4] and similar to that found in the work by Gordon, Fournet, and Maffeis [23, 24]. This language, Aura$_0$, is intended to model the fragment of Aura relevant to auditing. We show how proof-theoretic properties such as subject reduction and normalization can play a useful rôle in this context. One significant contribution of this paper is the normalization result for Aura$_0$ authorization proofs.

Section 4 gives an extended example of a file system interface that demonstrate that our techniques can be used to implement a reference monitor kernel such that any well-typed (but otherwise untrusted) client that satisfies the interface is forced to provide appropriate logging information. This approach is general in the sense that it allows the application developer to create domain-specific authorization policies even though the kernel interface is fixed. This example also show how this technique reduces the size of the trusted computing base and allows for the debugging of rules meant to capture particular domain-specific policies.

Of course there are many additional engineering problems that must be overcome before proof-enriched auditing becomes practical. Although it is not our intent to address all of those issues here, Section 5 highlights some of the salient challenges and sketches some future research directions. Section 6 discusses related work.

2 Kernel Mediated Access Control

A common system design idiom protects resources with a reference monitors [16]. The reference monitor takes requests from (generally) untrusted clients and may decide to allow or deny them. A well implemented reference monitor should be configured using a well-specified set of *rules* that define the current policy. Ideally, this collection of rules should mirror the relevant institutional policy intent.

Unfortunately, access control decisions may are not always made in accordance with institutional intent. This can occur for a variety of reasons including the following:

1. The reference monitor implementation or rule language may be insufficient to express institutional intent. It this case, some access decisions will necessarily be too restrictive or too permissive.

2. The reference monitor may be configured with an incorrect set of rules.

3. The reference monitor itself may be buggy. That is, it may reach an incorrect decision even when starting from correct rules.

Points (1) and (2) illustrate an interesting tension: rule language expressiveness is both necessary and problematic. While overly simple languages prevent effective realization of policy intent, expressive languages make it more likely that a particular rule set has unintended consequences. The latter issue is particularly acute in light of Harrison and colleagues' observation that determining the ultimate effect of policy changes in even simple systems is generally undecidable[27]. Point (3) recognizes that reference monitors may be complex, and likely to be vulnerable to implementation flaws.

The Aura programming model suggests a different approach to protecting resources. This is illustrated in Figure 1. There are three main components in the system: a trusted kernel, an untrusted application, and a set of rules that constitute the formal policy. The kernel itself contains a log and a resource to be protected. The application may only request resource access through kernel interface I_K. This interface (the op_is) wraps each of the resource's native operations, the $\mathsf{raw\text{-}op}_i$s in the figure, with new operations taking an additional argument, which is a proof that access is permitted. Σ_K and Σ_{ext} contain constant symbols that may be occur in these proofs.

Unlike a standard reference monitor, when given any request, the kernel always logs it and forwards the request to

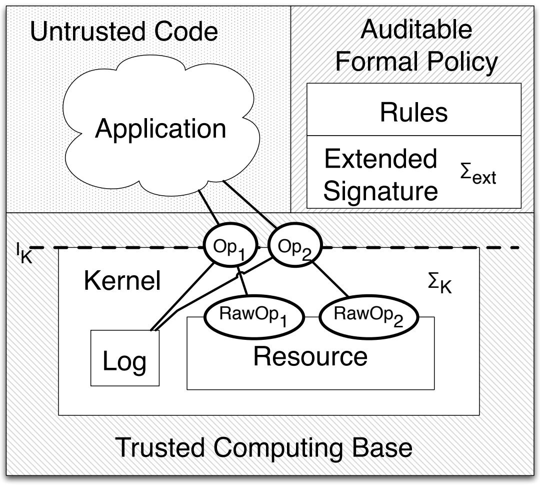

Figure 1. A monolithic application decomposed into several components operating with various degrees of trust.

the resource. How is the resource access mediated? The formal rules are expressed in a policy logic, and, in order to make well-typed calls to the kernel interface, the application must present a proof that access is permitted. While that application is untrusted, we assume it is well typed; thus any interface call contains an appropriate proof. Of course, this assumption may be weakened by requiring that each operations, op_is, perform dynamic proof—i.e. type—checking.

We define a language Aura$_0$ to provide an expressive policy logic for writing rules and kernel interface types. It is a cut-down version of full Aura, which itself is a polymorphic and dependent extension of Abadi's Dependency Core Calculus [5, 4].

Explicit in Aura$_0$ are notions of principals and assertions. Software components may be associated with one or more principals. Typically the trusted kernel will be identified with principal K, and the untrusted application may work on behalf of several principals: A, B, etc. Principals can state assertions; for instance the rule, or proposition, "the kernel asserts that all principals may 'write' any string," is written K **says** $((A{:}\textbf{prin}) \rightarrow (x{:}\textbf{string}) \rightarrow \mathsf{OkToWrite}\ A\ x)$. Evidence for such a rule will contain one or more signature objects—possibly, but not necessarily, cryptographic digital signatures—which irrefutably tie principals to their utterances.

The above design carries several benefits. Kernels log the reasoning used to reach access control decisions. Thus if a particular access control decision violates policy intent but is allowed by the rules, audit can reveal which rules contributed to this failure. Additionally, because all resource access is accompanied by a proof, the trusted computing base is limited to the proof checker and kernel. As small, standard programs these components are less likely to suffer from software vulnerabilities than ad-hoc reference monitors.

A key design principle is that kernels should small and general; this is realized by removing complex, specialized reasoning about policy (e.g. proof search) from the trusted computing base. In this sense, Aura systems are to traditional reference monitors as operating system microkernels are to monolithic kernels.

The formal computation model We model a system with three principle components: a trusted kernel K, an untrusted application a, and a security-oblivious resource R. The kernel mediates between the application and resource by checking and logging access control proofs.

A resource, R, is a stateful object with a set of operators that may query and update the state. Formally, $R = (\sigma, \mathit{States}, I_R)$ where

$$I_R = \mathsf{raw\text{-}op}_1 : T_1 \Rightarrow S_1, \ldots, \mathsf{raw\text{-}op}_n : T_n \Rightarrow S_n$$

The $\sigma \in \mathit{States}$ component is an arbitrary structure that representing R's current state, and *States* is the set of all possible resource states. I_R is the resource's interface; each $\mathsf{raw\text{-}op}_i : T_i \Rightarrow S_i$ is an operator with its corresponding type signature. Every operator has an interpretation as a total function with shape $[\![\mathsf{raw\text{-}op}_i]\!] : [\![T_i]\!] \times \mathit{States} \rightarrow [\![S_i]\!] \times \mathit{States}$ where $[\![T]\!]$ is the set of T's inhabitants. The input and output state arguments encode effectful update.

The kernel is the trusted system component that encapsulates and mediates access to a resource and maintains a log. For each raw operation $\mathsf{raw\text{-}op}_i$, the kernel interface provides a new logged version, op_i.

Formally a kernel is tuple $K = (L, R, \Sigma_K, I_K)$. The first component, L, is a list of proofs representing the log. The second component is the resource encapsulated by the kernel. Signature Σ_K contains pairs of predicates, $\mathsf{OkToOp}_i : T_i \rightarrow \textbf{Prop}$ and $\mathsf{DidOp}_i : T_i \rightarrow S_i \rightarrow \textbf{Prop}$ for each $\mathsf{raw\text{-}op}_i$ of type $T_i \Rightarrow S_i$ in R. These predicates serve as the core lexicon for composing access control rules: a proof of K **says** $\mathsf{OkToOp}\ t$ signifies that an operation is permitted with input t, and a proof of K **says** $\mathsf{DidOp}\ t\ s$ means that the the operation was run with input t and returned s. Lastly, the kernel exposes an application programming interface, I_k, which contains a lifted operation,

$$\mathsf{op}_i : (x : T_i) \Rightarrow \mathsf{K}\ \textbf{says}\ (\mathsf{OkToOp}_i\ x) \Rightarrow \\ \{y{:}S_i; \mathsf{K}\ \textbf{says}\ \mathsf{DidOp}_i\ x\ y\}$$

for each $\mathsf{raw\text{-}op}_i$ in R.

The type of op_i shows that the kernel expects two (curried) arguments before providing access to $\mathsf{raw\text{-}op}_i$. The first argument is simply $\mathsf{raw\text{-}op}_i$'s input. The second argument is a proof that the kernel approves the operation. Such a proof could be created by the kernel directly or, more typically, by the application composing proofs from *Rules*. The return value of op_i is a pair of $\mathsf{raw\text{-}op}_i$'s output with a proof that acts as a receipt, attesting that the kernel called $\mathsf{raw\text{-}op}_i$ and linking the call's input and output. Note that the propositions OkToOp_i and DidOp_i depend on the arguments x and y.

We implement operations in I_K by wrapping each underlying raw-op with trusted logging code. Each lifted operation is defined as follows:

```
1  $[\![\mathsf{op}_i]\!] = \lambda v.\lambda p.$
2    do  logAppend($p$)
3    let  $s = [\![\mathsf{raw\text{-}op}_i]\!]\ v$
4    let  $p' = \mathbf{sign}(\mathsf{K}, \mathsf{DidOp}_i\ v\ s)$
5    do  logAppend($p'$)
6    return  $\langle s, p' \rangle$
```

The total function $[\![\mathsf{op}_i]\!]$ takes two inputs[2]: a term v (representing the data needed by the operation) and a proof p. It returns a pair $\langle s, p' \rangle$ whose first component is a data value s and whose second component is a proof p' generated by the kernel to provide evidence that the operation was performed. We assume that all application calls to op_i respect this signature; thus the proof recorded in the log on line 2 must inhabit K **says** $\mathsf{OkToOp}_i\ v$. Lines 3 and 4 call into the underlying resource and construct a signature object attesting that this call occurred. Line 5 records the newly created proof.

The final components in the model are the application, the rule set, and the extended signature. The application is a program assumed to be well-typed, in that it respects the interface's type signature. The rule set is simply a well-known set of proofs. Lastly, the extended signature (Σ_{ext} in Figure 1) is a type signature which extends Σ_K.

Remote Procedure Call Example We will now examine a small example in our computation model. Consider the simple remote procedure call resource with only a single raw operation $\mathsf{raw\text{-}rpc} : \mathbf{string} \Rightarrow \mathbf{string}$. The kernel associated with this resource exposes the following predicates:

$$\begin{aligned}\Sigma_K = \ & \mathsf{OkToRPC} : \mathbf{string} \to \mathbf{Prop}, \\ & \mathsf{DidRPC} : \mathbf{string} \to \mathbf{string} \to \mathbf{Prop}\end{aligned}$$

and the kernel interface

$$\begin{aligned}I_K = \mathsf{rpc} : (x : \mathbf{string}) \Rightarrow \mathsf{K}\ \mathbf{says}\ \mathsf{OkToRPC}\ x \Rightarrow \\ \{y{:}\mathbf{string}; \mathsf{K}\ \mathbf{says}\ \mathsf{DidRPC}\ x\ y\}.\end{aligned}$$

[2]We abuse notation and treat pure mathematical functions logAppend and $[\![\mathsf{rawOp}_i]\!]$ as imperative procedures.

Terms

t	$::=$	$\mathbf{Kind}^P \mid \mathbf{Kind}^T$	Sorts
	$\mid$	$\mathbf{Prop} \mid \mathbf{Type}$	Kinds
	$\mid$	$\mathbf{string} \mid \mathbf{prin}$	Base Types
	$\mid$	$x \mid a \mid$	Variables and Constants
	$\mid$	$t\ \mathbf{says}\ t$	Says modality
	$\mid$	$(x{:}t) \to t$	Implication/Quantification
	$\mid$	$\{x{:}t; t\}$	Dependent Pair Type
	$\mid$	$\texttt{"a"} \mid \texttt{"b"} \mid \ldots$	String Literals
	$\mid$	$\mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \ldots$	Principal Literals
	$\mid$	$\mathbf{sign}(A, t)$	Signature
	$\mid$	$\mathbf{return}@[t]\ t$	Injection into **says**
	$\mid$	$\mathbf{bind}\ x = t\ \mathbf{in}\ t$	Reasoning under **says**
	$\mid$	$\lambda x{:}t.\, t \mid t\ t$	Abstraction and Application
	$\mid$	$\langle t, t \rangle$	Pair
	$\mid$	$(x{:}t) \Rightarrow t$	Computation Type

Figure 2. Syntax of Aura$_0$

One trivial policy permits any remote procedure call. This policy is most simply realized by the singleton rule set *Rules* $= \{r_0 : \mathsf{K}\ \mathbf{says}\ ((x{:}\mathbf{string}) \to \mathsf{OkToRPC}\ x)\}$.

3 The Logic

This section defines Aura$_0$, a language for expressing access control. Aura$_0$ is a higher-order, dependently typed, cut-down version of Abadi's Dependency Core Calculus [5, 4]. Following the Curry-Howard isomorphism [20], Aura$_0$ types correspond to propositions in access control logic, and expressions correspond to proofs. Dependent types allow propositions to be parameterized by objects of interest, such as principals or file handles. The interface between application and kernel code is defined using this language.

This section is organized as follows. First, we define the syntax and typing rules of Aura$_0$, followed by a few simple examples to illustrate how to use our language for access control. We then present the reduction rules of our language and discuss the importance of normalization with regard to auditing. Lastly, we state the formal properties of Aura$_0$.

3.1 Syntax

Figure 2 defines the syntax of Aura$_0$. There are two kinds of terms: propositions, which are used to construct access control proofs and are classified by types of the kind **Prop**, and expressions, which are classified by types of the kind **Type**. For ease of the subsequent presentation of the typing rules, we introduce two different sorts, $\mathbf{Kind}^P$ and $\mathbf{Kind}^T$, which classify **Prop** and **Type** respectively. Base types are **prin**, which is inhabited by principals, and **string**.

We use x to range over variables, and a to range over constants. String literals are `""`–enclosed ASCII symbols, and we use capital letters A, B, C etc. to denote literal principals. In addition to the standard constructs for the functional dependent type $(x{:}t_1) \to t_2$, dependent product type $\{x{:}t_1; t_2\}$, lambda abstraction $\lambda x{:}t_1.\, t_2$, function application $t_1\ t_2$, and dependent pair $\langle t_1, t_2 \rangle$, our language includes a special functional dependent type $(x{:}t_1) \Rightarrow t_2$. The distinction between $(x{:}t_1) \to t_2$ and $(x{:}t_1) \Rightarrow t_2$ lies in their typing judgments, and will be made clear in the next section. Intuitively, the distinction rules out complex computation at the type level. We will sometimes write $t_1 \to t_2$, $t_1 \Rightarrow t_2$, and $\{t_1; t_2\}$ as a shorthand for $(x{:}t_1) \to t_2$, $(x{:}t_1) \Rightarrow t_2$, and $\{x{:}t_1; t_2\}$ respectively when x does not appear free in t_2.

As in DCC, we use the modality **says** to represent access control logic formulas. The expression **return**$@[t_1]\ t_2$ is the introduction proof term for the type t_1 **says** t_2. The elimination term for the type t_1 **says** t_2 is the expression **bind** $x = t_1$ **in** t_2. Intuitively, the bind expression lets us use the proof t_1 in the proof t_2.

Finally, we use the expression **sign**(A, P) to represent a "signed" assertion. Such assertions are indisputable evidence of a principal's statement. By signing P—creating a proof of A **says** P—principal A expresses her belief that proposition P is true. Intuitively, assertions are verifiable, binding (i.e. non-repudiatable), and unforgeable. Implementation strategies are discussed in Section 5.

3.2 Type System

This section defines Aura$_0$'s type system. In the typing judgments, Σ denotes a signature that maps constants to their types. Γ denotes a variable typing context. Constants are considered globally fixed; variables only have meaning within local scopes. The formal definitions follow.

$$\begin{array}{lrcl} \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : t \\ \text{Signatures} & \Sigma & ::= & \cdot \mid \Gamma, a : t \end{array}$$

The four main typing judgments are:

Signature Typing	$\Sigma \vdash \diamond$	Signature Σ is well-formed
Context Typing	$\Sigma \vdash \Gamma$	Context Γ is well formed under signature Σ
Term Typing	$\Sigma; \Gamma \vdash t_1 : t_2$	Term t_1 has type t_2 under signature Σ, context Γ
Special Term Typing	$\Sigma; \Gamma \vdash t$	Term t is well-typed under signature Σ, context Γ

The typing rules for signatures and contexts are standard. The signature Σ is well-formed if Σ maps constants to types of sort $\mathbf{Kind}^P$. Thus all Aura$_0$ constants construct propositions. The context Γ is well-formed with respect to signature Σ if Γ maps variables to well-formed types.

A summary of the typing rules for terms can be found in Figure 3. Most of the rules are straightforward, and we explain only a few key rules. The rule T-SIGN states that a signed assertion created by the principal A signing a proposition P has type A **says** P. Since P can be any proposition, even false, sometimes certificates are also referred to as "false assertions". In other words, there is no justification for proposition P within the logic, except that principal A expressed her belief in P by signing it. These signed assertions are an essential part of encoding access control. The premises of T-SIGN typechecks A and P in the empty variable context, as signatures are intended to have unambiguous meaning in any scope—a signature with free variables is inherently meaningless.[3]

The rule T-RETURN states that if we can construct a proof term t_2 for proposition s_2, then the term **return**$@[t_1]\ t_2$ is a proof term for proposition t_1 **says** s_2. The T-BIND rule is a standard bind rule for monads. It states that what a principal A believes can only be used to deduce other propositions that A believes.

The rule for the functional dependent type T-PI restricts the kinds of dependencies allowed in our type system. Essentially, it rules out functional dependencies for the kind **Type**. Notice that in the T-LAM rule, the type of the lambda abstraction can only be of type **Prop**. These two rules allows us to express flexible access control rules. At the same time, it is rather straightforward to see that the restrictions on these rules eliminate type level computations, thus ensuring the decidability of type checking.

To express the interfaces between the application code and the kernel, however, we need more general forms of dependent types. To that end Aura$_0$ includes dependent product types and a special functional dependent type $x{:}t_1 \Rightarrow t_2$. Notice that dependent products have an introduction proof term but no elimination form. This is because we use the product dependent types only to associate data with the proofs that depend on these data.

The typing judgments for the special functional dependent type are at the bottom of Figure 3. The formulation of the rule is very similar to the typing judgment for $(x{:}t_1) \to t_2$, but the argument's type must itself have kind **Type** or **Prop** and the result type's kind is unrestricted. We could have relaxed the T-PI rule to include these cases, but we make this distinction for ease of reasoning about termination behavior at the type level.

3.3 Aura$_0$ Examples

The combination of dependent types and the says modality in Aura$_0$ can express many interesting policies. For in-

[3]The signature in line 4 of the definition $[\![\mathsf{Op}_i]\!]$ contains free variables, but this is acceptable because $[\![\mathsf{Op}_i]\!]$ is a mathematical function, not a program. The full Aura language can express such computations in programs.

$\boxed{\Sigma; \Gamma \vdash t : t}$

$$\frac{\Sigma \vdash \Gamma}{\Sigma; \Gamma \vdash \mathbf{Prop} : \mathbf{Kind}^P}\ \textsc{T-Prop} \qquad \frac{\Sigma \vdash \Gamma}{\Sigma; \Gamma \vdash \mathbf{Type} : \mathbf{Kind}^T}\ \textsc{T-Type} \qquad \frac{\Sigma \vdash \Gamma}{\Sigma; \Gamma \vdash \mathbf{string} : \mathbf{Type}}\ \textsc{T-string}$$

$$\frac{\Sigma \vdash \Gamma}{\Sigma; \Gamma \vdash \mathbf{prin} : \mathbf{Type}}\ \textsc{T-prin} \qquad \frac{\Sigma \vdash \Gamma \qquad x : t \in \Gamma}{\Sigma; \Gamma \vdash x : t}\ \textsc{T-var} \qquad \frac{\Sigma \vdash \Gamma \qquad a : t \in \Sigma}{\Sigma; \Gamma \vdash a : t}\ \textsc{T-const}$$

$$\frac{\Sigma; \Gamma \vdash t_1 : \mathbf{prin} \qquad \Sigma; \Gamma \vdash t_2 : \mathbf{Prop}}{\Sigma; \Gamma \vdash t_1 \ \mathbf{says}\ t_2 : \mathbf{Prop}}\ \textsc{T-says}$$

$$\frac{\Sigma; \Gamma \vdash t_1 : k_1 \qquad \Sigma; \Gamma, x : t_1 \vdash t_2 : k_2 \qquad k_1 \in \{\mathbf{Kind}^P, \mathbf{Type}, \mathbf{Prop}\} \qquad k_2 \in \{\mathbf{Kind}^P, \mathbf{Prop}\}}{\Sigma; \Gamma \vdash (x{:}t_1) \rightarrow \ t_2 : k_2}\ \textsc{T-pi}$$

$$\frac{\Sigma; \Gamma \vdash t_1 : k \qquad \Sigma; \Gamma, x : t_1 \vdash t_2 : k_2 \qquad k_1, k_2 \in \{\mathbf{Type}, \mathbf{Prop}\}}{\Sigma; \Gamma \vdash \{x{:}t_1; t_2\} : k_2}\ \textsc{T-sigma} \qquad \frac{\Sigma \vdash \Gamma \qquad s \in \{\texttt{"a"}, \texttt{"b"}, \ldots\}}{\Sigma; \Gamma \vdash s : \mathbf{string}}\ \textsc{T-litStr}$$

$$\frac{\Sigma \vdash \Gamma \qquad A \in \{\mathsf{A}, \mathsf{B} \ldots\}}{\Sigma; \Gamma \vdash A : \mathbf{prin}}\ \textsc{T-litPrin} \qquad \frac{\Sigma \vdash \Gamma \qquad \Sigma; \cdot \vdash t_1 : \mathbf{prin} \qquad \Sigma; \cdot \vdash t_2 : \mathbf{Prop}}{\Sigma; \Gamma \vdash \mathbf{sign}(t_1, t_2) : t_1 \ \mathbf{says}\ t_2}\ \textsc{T-sign}$$

$$\frac{\Sigma; \Gamma \vdash t_1 : \mathbf{prin} \quad \Sigma; \Gamma \vdash t_2 : s_2 \quad \Sigma; \Gamma \vdash s_2 : \mathbf{Prop}}{\Sigma; \Gamma \vdash \mathbf{return}@[t_1]\ t_2 : t_1 \ \mathbf{says}\ s_2}\ \textsc{T-return}$$

$$\frac{\Sigma; \Gamma \vdash e_1 : t \ \mathbf{says}\ P_1 \quad \Sigma; \Gamma, x : P_1 \vdash e_2 : t \ \mathbf{says}\ P_2}{\Sigma; \Gamma \vdash \mathbf{bind}\ x \ = \ e_1 \ \mathbf{in}\ e_2 : t \ \mathbf{says}\ P_2}\ \textsc{T-bind}$$

$$\frac{\Sigma; \Gamma, x : t \vdash p : P \qquad \Sigma; \Gamma \vdash (x{:}t) \rightarrow \ P : \mathbf{Prop}}{\Sigma; \Gamma \vdash \lambda x{:}t.\, p : (x{:}t) \rightarrow \ P}\ \textsc{T-lam} \qquad \frac{\Sigma; \Gamma \vdash t_1 : (x{:}P_2) \rightarrow \ P \qquad \Sigma; \Gamma \vdash t_2 : P_2}{\Sigma; \Gamma \vdash t_1 \ t_2 : \{t_2/x\}P}\ \textsc{T-app}$$

$$\frac{\Sigma; \Gamma \vdash t_1 : s_1 \qquad \Sigma; \Gamma \vdash t_2 : \{t_1/x\}s_2 \qquad \Sigma; \Gamma, x : s_1 \vdash s_2 : k}{\Sigma; \Gamma \vdash \langle t_1, t_2 \rangle : \{x{:}s_1; s_2\}}\ \textsc{T-pair}$$

$\boxed{\Sigma; \Gamma \vdash t}$

$$\frac{\Sigma; \Gamma \vdash t_1 : t_2 \quad t_2 \in \{\mathbf{Kind}^P, \mathbf{Kind}^T, \mathbf{Prop}, \mathbf{Type}\}}{\Sigma; \Gamma \vdash t_1}\ \textsc{T-C} \qquad \frac{\Sigma; \Gamma \vdash t_1 : k \quad k \in \{\mathbf{Type}, \mathbf{Prop}\} \quad \Sigma; \Gamma, x : t_1 \vdash t_2}{\Sigma; \Gamma \vdash (x{:}t_1) \ \Rightarrow \ t_2}\ \textsc{T-pi-C}$$

Figure 3. The typing relation

stance, Abadi's encoding of speaks-for [4] is easily adopted:

$$A \ \mathsf{speaksfor}\ B \triangleq B \ \mathbf{says}\ ((P{:}\mathbf{Prop}) \rightarrow \ A \ \mathbf{says}\ P \rightarrow P)$$

Adding dependency allows for more fine grained delegation. For instance we can encode partial delegation.

$$\mathsf{B}\ \mathbf{says}\ ((x{:}\mathbf{string}) \rightarrow \ \mathsf{A}\ \mathbf{says}\ \mathsf{Good}\ x \rightarrow \mathsf{Good}\ x)$$

Here A speaks-for B, but only when certifying that a string is "good." Such fine-grained delegation appears to be important in real applications where the full speaks-for relation is too permissive.

Recall the Remote Procedure Call example from Section 2. While an application might use r_0 directly when building proofs, it could also store a more convenient derived rule. This requires using Aura$_0$'s monadic bind to reason from K's perspective. For instance:

$$\begin{array}{lcl} r'_0 & : & (x{:}\textbf{string}) \to \ \mathsf{K}\ \textbf{says}\ \mathsf{OkToRPC}\ x \\ r'_0 & = & \lambda x{:}\textbf{string}.\ \textbf{bind}\ y\ =\ r_0\ \textbf{in return}@[\mathsf{K}] y\ x \end{array}$$

However, rules like r_0 and its derivatives are too trivial to admit interesting opportunities for audit.

A slightly more interesting policy says that any principal may perform a remote procedure call, so long as that principal signs the input string. One encoding uses the extended context

$$\Sigma_{ext} = \mathsf{ReqRPC} : \textbf{string} \to \textbf{Prop}, \Sigma_K.$$

and singleton rule set

$$\begin{array}{l} \mathit{Rules} = \{ r_1 = \ \textbf{sign}(\mathsf{K}, (x{:}\textbf{string}) \to \ (A{:}\textbf{prin}) \to \\ \qquad\qquad (A\ \textbf{says}\ \mathsf{ReqRPC}\ x) \to \mathsf{OkToRPC}\ x)\}. \end{array}$$

Given this rule, and auditor might find the following proofs in the log.

$$\begin{array}{l} p_1 = \ \textbf{bind}\ x\ =\ r_1 \\ \qquad \textbf{in return}@[\mathsf{K}](x\ \texttt{"hi"}\ \mathsf{A}\ \textbf{sign}(\mathsf{A}, \mathsf{ReqRPC}\ \texttt{"hi"})) \\ p_2 = \ (\lambda x{:}\mathsf{K}\ \textbf{says}\ \mathsf{OkToRPC}\ \texttt{"ab"}. \\ \qquad \lambda y{:}\mathsf{C}\ \textbf{says}\ \mathsf{ReqRPC}\ \texttt{"cd"}.\, x) \\ \qquad (\textbf{bind}\ z\ =\ r_1 \\ \qquad\quad \textbf{in return}@[\mathsf{K}](z\ \texttt{"ab"}\ \mathsf{B}\ \textbf{sign}(\mathsf{B}, \mathsf{ReqRPC}\ \texttt{"ab"})) \\ \qquad (\textbf{sign}(\mathsf{C}, \mathsf{ReqRPC}\ \texttt{"cd"})). \end{array}$$

We see that the first proof, p_1, contains A's signature. As signatures are unforgeable, the auditor can conclude that A is—in an informal, institutional policy sense—responsible for the request. Proof p_2 is more complicated; it contains signatures from both B and C. An administrator can learn several things from this proof.

We analyze p_2 using a reduction relation that simplifies proofs and which is defined in the following section. Taking the normal form of (i.e. simplifying) p_2 yields

$$\begin{array}{l} \textbf{bind}\ z\ =\ r_1 \\ \quad \textbf{in return}@[\mathsf{K}](z\ \texttt{"ab"}\ \mathsf{B}\ \textbf{sign}(\mathsf{B}, \mathsf{ReqRPC}\ \texttt{"ab"}). \end{array}$$

This term containing only B's signature: hence B may be considered accountable for the action. The inclusion of C's signature may be significant, however. If the application is intended to pass normal proofs to the kernel, this is a sign the application is malfunctioning. If principals are only supposed to sign certain statements, C's seemingly spurious signature may indicate a violation of rules outside the scope of Aura_0 on C's part.

$\boxed{\vdash t \to t'}$

$$\frac{x \notin \mathit{fv}(t_2)}{\vdash \textbf{bind}\ x\ =\ t_1\ \textbf{in}\ t_2 \to t_2}\ \text{R-BINDS}$$

$$\frac{}{\vdash \textbf{bind}\ x\ =\ \textbf{return}@[t_0]\ t_1\ \textbf{in}\ t_2 \to \{t_1/x\}t_2}\ \text{R-BINDT}$$

$$\frac{\vdash t_2 \to t'_2}{\vdash \textbf{return}@[t_1]\ t_2 \to \textbf{return}@[t_1]\ t_2}\ \text{R-SAYS}$$

$$\frac{y \notin \mathit{fv}(t_3)}{\begin{array}{c}\vdash \textbf{bind}\ x\ =\ (\textbf{bind}\ y\ =\ t_1\ \textbf{in}\ t_2)\ \textbf{in}\ t_3 \to \\ \textbf{bind}\ y\ =\ t_1\ \textbf{in bind}\ x\ =\ t_2\ \textbf{in}\ t_3\end{array}}\ \text{R-BINDC}$$

Figure 4. Selected reduction rules

3.4 Formal Properties of Aura_0

Subject reduction As the preceding example illustrates, proof simplification is a useful tool for audit. Following the Curry-Howard isomorphism, proof simplification corresponds to λ-calculus reductions on proof terms.

We present selected reduction rules for our language in Figure 4. A full set of the reduction rules can be found in the Appendix. Most of the reduction rules are standard. For the bind expression, in addition to the standard congruence, beta reduction, and commute rules as we often see in monadic languages, we also include a special beta reduction rule T-BINDS. The T-BINDS rule eliminates bound proofs which are never mentioned in the bind's body. Note that the reduction rules also include beta reduction under the says monad, which is not present in standard monadic languages. The reason for this is that in a monadic setting, the computation trapped under the monad may contain effects and thus must be evaluated lazily. In the access control logic setting, however, such effects are not a concern. Note also that there is no reduction under signatures or **says**. Intuitively, this is because signatures represent fixed objects realized in cryptography or a similarly immutable form. Allowing reductions under signatures and **says** does not affect the soundness of the system, but doing so complicates the proof of strong normalization and implementation strategies.

The following lemma states that the typing of an expression is preserved under reduction rules.

Lemma 3.1 (Subject Reduction)**.** *If* $\vdash t \to t'$ *and* $\Sigma; \Gamma \vdash t : s$ *then* $\Sigma; \Gamma \vdash t' : s$.

Proof Sketch. Proof is by structural induction on the reduction relation. It requires several standard facts. Additionally, the R-BINDS cases requires a non-standard lemma saying that we may remove a variable x from the typing context when x is not used elsewhere in the typing judgment. $\square$

Proof normalization We say an expression is in normal form if there are no applicable reduction rules. As we have shown in the example above, given a proof term t for proposition P, it is useful to reduce t to a normal form. We call this process proof normalization. In order for proof normalization to be viable, we need to know (1) whether the normalization process will terminate, and (2) whether the normal form is unique. The answer to both of these questions, in this case, is yes.

We say an expression t is strongly normalizing if application of any sequence of reduction rules to t always terminates. We say a language is strongly normalizing if all the terms in the language are strongly normalizing. We have proved that Aura$_0$ is strongly normalizing, which implies that any algorithm for proof normalization will terminate. The details of the proofs are presented in the Appendix A.3. Here we present only a sketch of the proof.

Lemma 3.2 (Strong Normalization)**.** *Aura$_0$ is strongly normalizing.*

Proof Sketch. We prove Aura$_0$ is strongly normalizing by translating Aura$_0$ to the Calculus of Construction extended with product dependent types, which is known to be strongly normalizing [26]. The key property of the translation is that it preserves types and reduction steps. The interesting cases are the translations of DCC terms. The translation drops the **says** monad, and translates the **bind** expression to lambda application. The term $\textbf{sign}(t_1, t_2)$ has type t_1 **says** t_2 and is translated to a variable whose type is the translation of t_2. One subtlety of the proof is tracking the dependency in the types of these introduced variables, which must be handled delicately. $\square$

We have also proved that Aura$_0$ is confluent.

Lemma 3.3 (Confluence)**.** *If $t \to^* t_1$, and $t \to^* t_2$, then there exists t_3 such that $t_1 \to^* t_3$ and $t_2 \to^* t_3$.*

Proof Sketch. We first prove that Aura$_0$ is weakly confluence. We use the well-known result that if a language is strongly normalizing and has weak confluence property, then it is confluence. $\square$

A direct consequence is that the normal form of an expression is unique. Given an expression, any algorithm for proof normalization will yield the same normal form. Therefore, the set of relevant evidence (i.e., signatures) in a proof term is also unique.

4 File System Example

As a more concrete example, we consider a filesystem in which file access is authorized using our logic and log entries consist of these authorization proofs. In a traditional filesystem authorization decisions regarding file access are made when a file is opened, and thus we begin by considering only the open operation; additional operations are discussed in Section 4.1. Our open is intended to provide flexible access control on top of a system featuring a corresponding raw-open and associated constants:

$$\mathsf{Mode} : \textbf{Type} \qquad \mathsf{FileDes} : \textbf{Type}$$

$$\mathsf{RDONLY} : \mathsf{Mode} \qquad \mathsf{WRONLY} : \mathsf{Mode}$$
$$\mathsf{APPEND} : \mathsf{Mode} \qquad \mathsf{RDWR} : \mathsf{Mode}$$

$$\mathsf{raw\text{-}open} : \{\mathsf{Mode}; \textbf{string}\} \Rightarrow \mathsf{FileDes}$$

We can imagine that raw-open is part of the interface to an underlying filesystem with no notion of per-user access control or Aura$_0$ principals; it, of course, will not be exposed outside of the kernel. Taking inspiration from Unix, we define RDONLY, WRONLY, APPEND, and RDWR (the inhabitants of Mode), which specify whether to open a file for reading only, overwrite only, append only, or unrestricted reading and writing respectively. Left undefined is the type FileDes, which is assumed to be some sort of unforgeable capability—i.e., a file descriptor—that can be used to access the contents of an opened file.

Figure 5 shows interface to open, the extended signature of available predicates, and the rules used to construct the proofs of type K **says** OkToOpen $\langle f, m\rangle$ for some file f and mode m that open requires. OkToOpen and DidOpen are as specified in Section 2, and the other predicates have the obvious readings: Owns $A\ f$ states that the principal A owns the file f, ReqOpen $m\ f$ is a request to open file f with mode m, and Allow $A\ m\ s$ states that A should be allowed to open f with mode m. (As we are not modeling authentication we will take it as given that all proofs of type A **says** ReqOpen $m\ f$ come from A; we discuss ways of enforcing this in Section 5.) We assume, for each file f, the existence of a rule owner$f$ of type K **says** Owns A f for some constant principal A—as only one such rule exists for any f and no other means are provided to generate proofs of this type, we can be sure that each file will always have a unique owner. Aside from such statements of ownership, the only rule a filesystem absolutely needs is delegate, which states that the kernel allows anyone to access a file with a particular mode if the owner of the file allows it.

The other rules are of great convenience, however; owned relieves the file owner A from the need to create signatures of type A **says** Allow $A\ m\ f$ for files A owns, readwrite allows a user who has acquired read and write

$$
\begin{aligned}
\mathsf{OkToOpen} &: \{\mathsf{Mode}; \textbf{string}\} \rightarrow \textbf{Prop} \\
\mathsf{DidOpen} &: (x : \{\mathsf{Mode}; \textbf{string}\}) \rightarrow \\
&\quad \mathsf{FileDes} \rightarrow \textbf{Prop} \\[1em]
\mathsf{open} &: (x : \{\mathsf{Mode}; \textbf{string}\}) \Rightarrow \\
&\quad \mathsf{K}\ \textbf{says}\ \mathsf{OkToOpen}\ x \Rightarrow \\
&\quad \{h{:}\mathsf{FileDes}; \mathsf{K}\ \textbf{says}\ \mathsf{DidOpen}\ x\ h\} \\[1em]
\mathsf{Owns} &: \textbf{prin} \rightarrow \textbf{string} \rightarrow \textbf{Prop} \\
\mathsf{ReqOpen} &: \mathsf{Mode} \rightarrow \textbf{string} \rightarrow \textbf{Prop} \\
\mathsf{Allow} &: \textbf{prin} \rightarrow \mathsf{Mode} \rightarrow \textbf{string} \rightarrow \textbf{Prop} \\[1em]
\mathsf{owner}f &: \mathsf{K}\ \textbf{says}\ \mathsf{Owns}\ \mathsf{A}\ f \\[1em]
\mathsf{delegate} &: \mathsf{K}\ \textbf{says}\ ((A : \textbf{prin}) \rightarrow (B : \textbf{prin}) \rightarrow \\
&\qquad (m : \mathsf{Mode}) \rightarrow (f : \textbf{string}) \rightarrow \\
&\qquad A\ \textbf{says}\ \mathsf{ReqOpen}\ m\ f \rightarrow \\
&\qquad \mathsf{K}\ \textbf{says}\ \mathsf{Owns}\ B\ f \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ m\ f \rightarrow \\
&\qquad \mathsf{OkToOpen}\ m\ f) \\[1em]
\mathsf{owned} &: \mathsf{K}\ \textbf{says}\ ((A : \textbf{prin}) \rightarrow (m : \mathsf{Mode}) \rightarrow \\
&\qquad (f : \textbf{string}) \rightarrow \\
&\qquad A\ \textbf{says}\ \mathsf{ReqOpen}\ m\ f \rightarrow \\
&\qquad \mathsf{K}\ \textbf{says}\ \mathsf{Owns}\ A\ f \rightarrow \\
&\qquad \mathsf{OkToOpen}\ m\ f) \\[1em]
\mathsf{readwrite} &: \mathsf{K}\ \textbf{says}\ ((A : \textbf{prin}) \rightarrow (B : \textbf{prin}) \rightarrow \\
&\qquad (f : \textbf{string}) \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDONLY}\ f \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{WRONLY}\ f \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDWR}\ f) \\[1em]
\mathsf{read} &: \mathsf{K}\ \textbf{says}\ ((A : \textbf{prin}) \rightarrow (B : \textbf{prin}) \rightarrow \\
&\qquad (f : \textbf{string}) \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDWR}\ f \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDONLY}\ f) \\[1em]
\mathsf{write} &: \mathsf{K}\ \textbf{says}\ ((A : \textbf{prin}) \rightarrow (B : \textbf{prin}) \rightarrow \\
&\qquad (f : \textbf{string}) \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDWR}\ f \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{WRONLY}\ f) \\[1em]
\mathsf{append} &: \mathsf{K}\ \textbf{says}\ ((A : \textbf{prin}) \rightarrow (B : \textbf{prin}) \rightarrow \\
&\qquad (f : \textbf{string}) \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDWR}\ f \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{APPEND}\ f)
\end{aligned}
$$

Figure 5. Interface, extended signature, and rules for file access control

permission for a file from different sources to open the file for reading and writing simultaneously, and read, write, and append do the reverse, allowing a user to drop from RDWR mode to RDONLY, WRONLY, or APPEND. These last three rules simply reflect the semantic fact that read-write permission is equivalent to read-only permission combined with write-only permission.

Proof creation is done in the same manner as shown in the RPC example in the previous sections. For brevity we omit most proof terms in this section, as they can grow quite large, but a simple use of the rule owned by A to read the file `"foo.txt"` might appear as follows:

$$
\begin{aligned}
\mathsf{owner}f &: \mathsf{A}\ \textbf{says}\ \mathsf{ReqOpen}\ \mathsf{RDONLY}\ \texttt{"foo.txt"} \\
\textit{req} &= \textbf{sign}(\mathsf{A}, \mathsf{ReqOpen}\ \mathsf{RDONLY}\ \texttt{"foo.txt"})
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{bind}\ o\ =\ \mathsf{owned} \\
&\quad \textbf{in return}@[\mathsf{K}](o\ \mathsf{A}\ \mathsf{RDONLY}\ \texttt{"foo.txt"}\ \textit{req}\ \mathsf{owner}f)
\end{aligned}
$$

With the rules given in Figure 5 and the other constructs of our logic it is also easy to create complex chains of delegation for file access. For example, Alice (A) may delegate full authority over any files she can access to Bob (B) with a signature of type

$$
\begin{aligned}
&\mathsf{A}\ \textbf{says}\ (C : \textbf{prin} \rightarrow m : \mathsf{Mode} \rightarrow f : \textbf{string} \rightarrow \\
&\quad \mathsf{B}\ \textbf{says}\ \mathsf{Allow}\ C\ m\ f \rightarrow \mathsf{A}\ \textbf{says}\ \mathsf{Allow}\ C\ m\ f),
\end{aligned}
$$

or she may restrict what Bob may do by adding further requirements on C, m, or f. She might restrict the delegation to files that she owns, or replace C with B to prevent Bob from granting access to anyone but himself. Chains of reasoning constructed out of such user-created acts of delegation can grow to be arbitrarily long.

As described in Section 2, the kernel logs the arguments to our interface functions whenever they are called. So far we have only one such function, open, and logging its arguments means keeping a record every time the system permits a file to be opened. Given the sort of delegation chains that the rules allow, it should be clear that the reason why an open operation is permitted can be rather complex, which provides a strong motivation for the logging of proofs.

Logging file opens can also allow a system administrator to debug the rule set. The rules in Figure 5 might well be supplemented with, for example

$$
\begin{aligned}
r_1 &: \mathsf{K}\ \textbf{says}\ ((A : \textbf{prin}) \rightarrow (B : \textbf{prin}) \rightarrow \\
&\qquad (f : \textbf{string}) \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDONLY}\ f \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{APPEND}\ f \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDWR}\ f) \\[1em]
r_2 &: \mathsf{K}\ \textbf{says}\ ((A : \textbf{prin}) \rightarrow (B : \textbf{prin}) \rightarrow \\
&\qquad (f : \textbf{string}) \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{WRONLY}\ f \rightarrow \\
&\qquad B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{APPEND}\ f)
\end{aligned}
$$

Rule r_1 is clearly erroneous, as it allows a user with only permission to read from and append to a file to alter its existing content, but such a rule could easily be introduced by human error when the rule set is created. Since any uses of this rule would be logged, however, it would not be possible to exploit such a problematic rule without leaving a clear record of how it was done, allowing a more prudent administrator to correct the rule set.

Rule r_2, on the other hand, is a bit more subtle—it states that the ability to overwrite a file is strictly more powerful than the ability to append to that file, even in the absence of any ability to read. Whether such a rule is valid depends on the expectations of the system's users; r_2 is clearly unacceptable if users desire to allow others to overwrite but not append to files, yet if they do not, r_2 may be seen as quite convenient, allowing for, among other things, easy continuation of long write operations that were prematurely aborted. Examining the proofs in the log—especially if the log includes invalid proofs rejected at runtime, a possibility discussed in Section 5—can help the administrator determine whether the inclusion of r_2 best suits the needs of most users.

4.1 Extensions

We have so far discussed only open, but there is still much our logic has to offer a filesystem, even in the context of operations that do not involve authorization.

Reading and writing While access permission is granted when a file is opened, it is worth noting that, as presented, the type FileDes conveys no information about what sort of access has been granted; consequently, attempting, for example, to write to a read-only file descriptor will result in a run-time error. Since we already have a system with dependent types, this need not be the case; while it is somewhat orthogonal to our concerns of authorization, FileDes could easily be made to depend on the mode with which a file has been opened, and operations could expect file descriptors equipped with the correct Mode arguments. This would, however, require both some analog to the subsumption rules read, write, and append, as well as, for pragmatic reasons, a means of preventing the kernel from logging what is being written during file writes.

Close At first glance it seems that closing a file belongs with reading and writing as an operation that depends only on having a valid file descriptor to ensure success, yet there is something more we can gain from our types. For example, if we require a corresponding DidOpen when constructing proofs of type OkToClose, we can allow a user to share an open file descriptor with other processes secure in the knowledge that those processes will be unable to prematurely close the file. In addition, logging of file close operations can help pinpoint erroneous double closes, which, while technically harmless, may be signs of deeper logic errors in the program that triggered them.

Ownership File creation and deletion are certainly operations that have something to do with authorization, and they are especially interesting due to their interaction with the Owns predicate. The creation of file f by principal A should introduce a rule owner$f$: Owns A f into the rule set, while the deletion of a file should remove said rule. A means of transferring file ownership would also be desirable. This can amount to treating a subset of our rules as a protected resource in their own right, with a protected interface to these rules and further rules about when access to this new resource should be granted. An alternate approach is to dispense with ownership rules completely and instead use signed objects and signature revocation, discussed further in Section 5, to represent ownership.

5 Discussion

Signature implementation Thus far we've treated signatures as completely abstract. The key property we have asked of signatures is that they be unforgeable. This interface lends itself to two different implementations.

The first approach is cryptographic. A signature can be constructed by using public key cryptography. Each principal must be associated with a key pair, and the public keys should be well known. Implementing rule T-SIGN reduces to calling the signature scheme's verification function. The cryptographic scheme is well suited for distributed systems with mutual distrust.

A decision remains to be made, however: we can interpret **sign**(A, P) as a tuple containing the cryptographic signature and also A and P in plaintext, or we can interpret it as only the cryptographic signature. In the latter case signatures are small (potentially constructed from a hash of the contents), but recovering the text of a proposition from its proof (i.e. doing type inference) is computationally infeasible. In the former case, inference is trivial, but proofs are generally large. Note that proof checking **sign**s (that is, given a proof and a proposition, determining whether the proof supports the proposition) is a polynomial time operation in either case.

An alternative implementation of signatures assumes that all principals trust some party, *moderator*, who maintains a table of signatures as abstract data values. Each **sign** may then be represented as an index into the moderator's table. These proof handles can be made small, without complicating type inference, but such a moderated scheme requires a closed system with a mutually trusted component.

In a small system, this can be the kernel itself, but a larger system might contain several kernels protecting different resources and administered by disparate organizations, in which case finding a suitable moderator may become quite difficult, but the savings as compared with expensive public key cryptography may be worth the difficulty.

Temporary signatures Real-world digital signature implementations generally include with each signature an interval of time outside of which the signature should not be considered valid. In addition, there is often some notion of a revocation list to which signatures can be added to ensure their immediate invalidation. Both of these concepts could be useful in our setting, as principals might want to delegate authority temporarily and might or might not know in advance how long this delegation could last. Potentially mutable rules can even be represented by digital signatures—which could be very important in a truly distributed setting—in the presence of a revocation list.

The question remains, however, how best to integrate these concepts with the authorization logic. One possible answer is to change nothing in the logic and simply allow for the possibility that any proof could be declared invalid at runtime due to an expired signature. Following this strategy requires the log operations to dynamically validate the timestamps in the signatures before logging, thereby making the kernel operations partial (the validity checks might fail). In such a setting, it seems appealing to incorporate some kind of transaction mechanism so that clients can be guaranteed that their proofs are current before attempting to pass them to the kernel. While easy to implement, this approach may be unsatisfying in that programmers are left unable to reason about or account for such invalid proofs.

Signatures might also be limited in the number of times they may be used, and this seems like a natural application for *linear types* (see Bauer *et al.* for an authorization logic with linearity constraints [12]). Objects of a linear type must be used exactly once, making linear types appropriate for granting a user access to a resource only a set number of times. They can also be used to represent protocols at the type level, ensuring, for example, that a file descriptor is not used after it is closed.

Garg, deYoung, and Pfenning [22] are studying a constructive and linear access control logic with an explicit time intervals. Their syntax includes propositions of the form $P@[T_1, T_2]$, meaning "P is valid between times T_1 and T_2." Proofs relying on such proposition are only valid during at certain times. To handle this, the judgment system is parameterized by an interval; the interpretation of sequent $\Psi; \Gamma; \Delta \Longrightarrow P[I]$ is, "given assumptions Ψ, Γ, and Δ, P is valid during interval I." Adopting this technique could be a useful generalization of Aura$_0$ to address the problems of temporal policies, though it is currently unclear what representations of time and revocation might best balance concerns of simplicity and expressive power.

Authentication In Section 4 we assumed that signatures of type A **says** $\mathsf{ReqOpen}\ m\ f$ are always sent from A. Such an assumption is necessary because we are not currently modeling any form of authentication—or even the association of a principal with a running program—but a more realistic solution is needed when moving beyond the scope of this paper. For example, communication between programs running on behalf of different principals could take place over channel endpoints with types that depend on the principal on the other end of the channel.

Of course, when this communication is between different machines on an inherently insecure network, problems of secure authentication become non-trivial, as we essentially hope to implement a secure channel on top of an insecure one. In real life this is done through cryptography, and one of the long-term goals of the Aura project is to elegantly integrate these cryptographic methods with our type system.

Pragmatics We are in the process of implementing Aura, in part to gain practical experience with the methodology proposed in this paper. Besides the technical issues with temporal policies and authentication described above, we anticipate several other concerns that need to be addressed.

In particular, we will require efficient log operations and compact proof representations. Prior work on proof compression in the context of proof-carrying code [31] should apply in this setting, but until we have experience with concrete examples, it is not clear how large the authorization proofs may become in practice. A related issue is what kind of tool support is necessary for browsing and querying the audit logs. Such a tool should allow system administrators to issue queries against the log, manipulate the evidence present in the logs, and possibly receive help in debugging rule sets.

For client developers, we expect that it will often prove useful to log information beyond that which is logged by the kernel. A simple means of doing this is to treat the log object as a resource protected by the kernel. The kernel interface could expose a generic "log" operation

$$\mathsf{log} : (x : \textbf{string}) \to \mathsf{K}\ \textbf{says}\ \mathsf{OkToLog}\ x \to \mathsf{K}\ \textbf{says}\ \mathsf{DidLog}\ x$$

with (hopefully) permissive rules as to the construction of $\mathsf{OkToLog}$ proofs. As discussed in Section 4, it might prove especially useful to log failed attempts at proof construction. Conversely, some operations take arguments that should not be logged, for security or space constraints.

6 Related Work

Earlier work on proof-carrying access control [6, 8, 18, 13, 14, 23] recognized the importance of "says", but Abadi [4] was the first to define it as an indexed monad, as in Aura$_0$. Abadi *et al.*'s work [5] also proved DCC's key noninterference property: in the absence of delegation, nothing B says can cause A to say false. Aura$_0$ builds on DCC in several ways. The addition of dependent types enhances the expressiveness of DCC, and the addition of **sign** allows for a robust distributed interpretation of **says**. Aura$_0$'s treatment of principals as terms, as opposed to members of a special index set, enables quantification over principals. Lastly Aura$_0$ eliminates DCC's protects relation (which allows additional commutation and simplification of **says** types). Dropping the protects relation eliminates unintuitive DCC tautologies, such as $(\mathsf{A}\ \textbf{says}\ \mathsf{B}\ \textbf{says}\ P) \rightarrow (\mathsf{B}\ \textbf{says}\ \mathsf{A}\ \textbf{says}\ P)$, and ensures desired **says** manipulations are explicitly recorded in proofs.

Our work is closely related to Fournet, Gordon and Maffeis's [23, 24] research on authorization in distributed systems. Fournet *et al.* work with an explicit, π-calculus based model of computation. Like us they use dependent types to express access control properties. Fournet and colleagues focus on the security properties that are maintained during execution. These properties are reflected into the type system using static theorem proving and a type constructor, **Ok**. However, the inhabitants of **Ok** do not contain dynamic information and cannot be logged for later audit. Additionally, while Aura$_0$ treats signing abstractly, Fournet and colleagues' type system (and computation model) can explicitly discuss cryptographic operations.

Trust management systems like PolicyMaker and Keynote are also related [17]. Trust Management systems are intended to answer the question "Does the set C of credentials, prove that the request r complies with the local security policy P." [17] Such systems use general purpose compliance checkers to verify credentials. In PolicyMaker proofs are programs, written in a safe language, that operate on strings. Request r is allowed when the application can compose such programs such that the on input r the composed program returns true. Note that while validity of proposition in Aura$_0$ is tested by type checking, validity in PolicyMaker is tested by *evaluation*; these are fundamentally different approaches to logic. Similar to this paper, trust management systems intend for proof checking to occur in a small and general trusted computing base. Proof search may be delegated to untrusted components.

Proof Carrying Access Control has been field tested by Bauer and colleagues in the Grey project [13, 14]. In their project, smart phones build proofs which can be used to open office doors or log into computer systems. The Grey architecture shares structural similarities to the model discussed in this paper. In Grey, devices generating proofs, like our applications, need not reside in the trusted computing base. Additionally, both systems use expressive foundational logics to define policies, higher-order logic in the case of Grey [19]. In order to make proof search effective, Bauer suggests using cut-down fragments of higher order logic for expressing particular rule sets and using a distributed, tactic-based proof search algorithm.

Wee implemented the Logging and Auditing File System (LAFS) [34], a practical system that shares several architectural elements with our design. LAFS uses a lightweight daemon, analogous to our kernel, to wrap NFS file systems. Like our kernel, the LAFS daemon forwards all requests to the underlying resources. Both systems configure policy using sets of rules defined outside the trusted computing base. The systems differ in three key respects. First, the LAFS policy language is too weak to express many Aura$_0$ policies. Second, Aura$_0$ requires some privileged $\mathsf{K}\ \textbf{says}\ \cdot$ rules to bootstrap a policy; but LAFS can be completely configured with non-privileged policies. Third, the LAFS interface is designed to be transparent to application code, and does not provide any access control properties. Instead LAFS logs, but does not prevent, rules violations.

Cederquist and colleagues [18] describe a distributed system architecture with discretionary logging and no reference monitor. In this system agents (i.e. principals) may choose to enter proofs (written in a first-order natural deduction style logic) into a a trusted log when performing actions. Cederquist et. al. formalizes accountability, so that agents are guilty until proved innocent; that is, agents use log entries to reduce the quantity of actions they are accountable for. This relies on the ability of an authority to independently observe some actions. Such observations appear necessary to begin the audit process.

7 Conclusion

This paper has argued for evidence-based auditing, in which audit log entries contain proofs about authorization; such proofs are useful for minimizing the trusted computing base and provide information that can help debug policies. This paper presents an architecture for structuring systems in terms of trusted kernels whose interfaces require proofs. As a concrete instance of this approach, this paper has developed Aura$_0$, a dependently-typed authorization logic that enjoys subject reduction and strong normalization properties. Several examples using Aura$_0$ demonstrate how we envision applying these ideas in practice.

References

[1] Martín Abadi. Logic in access control. In *Proceedings of the 18th Annual Symposium on Logic in Computer*

Science (LICS'03), pages 228–233, June 2003.

[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 Transactions on Programming Languages and Systems*, 15(4):706–734, October 1993.

[4] Martn Abadi. Access control in a core calculus of dependency. *Electronic Notes in Theoretical Computer Science*, 172:5–31, April 2007.

[5] Martn Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, January 1999.

[6] Martn Abadi, Michael Burrows, Butler W. Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[7] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 52–62, November 1999.

[8] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *CCS '99: Proceedings of the 6th ACM conference on Computer and communications security*, pages 52–62, New York, NY, USA, 1999. ACM.

[9] S. Axelsson, U. Lindqvist, U. Gustafson, and E. Jonsson. An approach to UNIX security logging. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 62–75, 1998.

[10] Gilles Barthe, John Hatcliff, and Peter Thiemann. Monadic type systems: Pure type systems for impure settings (preliminary report). In Andrew Gordon, Andrew Pitts, and Carolyn Talcott, editors, *Proc. of 2nd Wksh. on Higher-Order Operational Techniques in Semantics, HOOTS'97, Stanford Univ., CA, USA, 8–11 Dec. 1997*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam, 1998.

[11] Lujo Bauer. *Access Control for the Web via Proof-Carrying Authorization*. PhD thesis, Princeton University, November 2003.

[12] Lujo Bauer, Kevin D. Bowers, Frank Pfenning, and Michael K. Reiter. Consumable credentials in logic-based access control. Technical Report CMU-CYLAB-06-002, Carnegie Mellon University, February 2006.

[13] Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *Information Security: 8th International Conference, ISC 2005*, volume 3650 of *Lecture Notes in Computer Science*, pages 431–445, September 2005.

[14] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security & Privacy*, pages 81–95, May 2005.

[15] Mihir Bellare and Bennet Yee. Forward integrity for secure audit logs. Technical report, Computer Science and Engineering Department, University of California at San Diego, November 1997.

[16] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2002.

[17] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet programming: security issues for mobile and distributed objects*, pages 185–210. Springer-Verlag, London, UK, 1999.

[18] J.G. Cederquist, R. Corin., M.A.C. Dekker, S. Etalle, and J.J. den Hartog. An audit logic for accountability. In *The Proceedings of the 6th IEEE International Workshop on Policies for Distributed Systems and Networks*, 2005.

[19] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[20] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[21] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113, May 2002.

[22] Henry deYoung, Deepak Garg, and Frank Pfenning. An authorization logic with explicit time, 2007. Draft, by personal communication.

[23] Cdric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In

Proc. of the 14th European Symposium on Programming, volume 3444 of *LNCS*, pages 141–156, Edinburgh, Scotland, April 2005. Springer-Verlag.

[24] Cdric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for distributed systems. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, pages 31–45, Venice, Italy, July 2007.

[25] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW 19)*, pages 283–296, 2006.

[26] Herman Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In *TYPES '94: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 14–38, London, UK, 1995. Springer-Verlag.

[27] M. A. Harrison, W. L Ruzzo, and J. D. Ullman. Protection in operating systems. *Comm. of the ACM*, 19(8):461–471, August 1976.

[28] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 31. IEEE Computer Society, 1997.

[29] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 175, Washington, DC, USA, 1997. IEEE Computer Society.

[30] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1):128–171, 2003.

[31] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, page 93, Washington, DC, USA, 1998. IEEE Computer Society.

[32] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th on USENIX Security Symposium*, pages 53–62, Berkeley, CA, USA, January 1998.

[33] B. Waters, D. Balfanz, G. E. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *11th Annual Network and Distributed Security Symposium (NDSS '04)*, San Diego, CA, USA, February 2004.

[34] Christopher Wee. LAFS: A logging and auditing file system. In *Annual Computer Security Applications Conference*, pages 231–240, New Orleans, LA, USA, December 1995.

A Appendix

A.1 Definitions

Definition A.1 (Metafunctions over syntax).

$$\begin{array}{rcl}
\{s/z\}z &=& s \\
\{s/z\}x &=& x \quad \textit{if } z \neq x \\
\{s/z\}(x{:}t_1) \rightarrow\ t_2 &=& (x{:}\{s/z\}t_1) \rightarrow\ \{s/z\}t_2 \quad \textit{if } z \neq x \\
\{s/z\}\{x{:}t_1; t_2\} &=& \{x{:}\{s/z\}t_1; \{s/z\}t_2\} \quad \textit{if } z \neq x \\
\{s/z\}\textbf{\textit{sign}}(t_1, t_2) &=& \textbf{\textit{sign}}(\{s/z\}t_1, \{s/z\}t_2) \\
\{s/z\}\textbf{\textit{return}}@[t_1]\ t_2 &=& \textbf{\textit{return}}@[\{s/z\}t_1]\ \{s/z\}t_2 \\
\{s/z\}\textbf{\textit{bind}}\ x\ =\ t_1\ \textbf{\textit{in}}\ t_2 &=& \textbf{\textit{bind}}\ x\ =\ \{s/z\}t_1\ \textbf{\textit{in}}\ \{s/z\}t_2 \quad \textit{if } z \neq x \\
\{s/z\}\lambda x{:}t_1.\, t_2 &=& \lambda x{:}\{s/z\}t_1.\, \{s/z\}t_2 \quad \textit{if } z \neq x \\
\{s/z\}t_1\ t_2 &=& (\{s/z\}t_1)\ (\{s/z\}t_2) \\
\{s/z\}\langle t_1, t_2\rangle &=& \langle \{s/z\}t_1, \{s/z\}t_2\rangle \\
\{s/z\}t &=& t \quad \textit{otherwise}
\end{array}$$

$$\begin{array}{rcl}
\{s/z\}\cdot &=& \cdot \\
\{s/z\}\Gamma, x : t &=& (\{s/z\}\Gamma), x : \{s/z\}t
\end{array}$$

$$\begin{array}{rcl}
\mathit{fv}(x) &=& \{x\} \\
\mathit{fv}(t_1\ \textbf{\textit{says}}\ t_2) &=& \mathit{fv}(t_1) \cup \mathit{fv}(t_2) \\
\mathit{fv}((x{:}t_1) \rightarrow\ t_2) &=& \mathit{fv}(t_1) \cup \mathit{fv}(t_2) \setminus \{x\} \\
\mathit{fv}(\{x{:}t_1; t_2\}) &=& \mathit{fv}(t_1) \cup \mathit{fv}(t_2) \setminus \{x\} \\
\mathit{fv}(\textbf{\textit{sign}}(t_1, t_2)) &=& \mathit{fv}(t_1) \cup \mathit{fv}(t_2) \\
\mathit{fv}(\textbf{\textit{return}}@[t_1]t_2) &=& \mathit{fv}(t_1) \cup \mathit{fv}(t_2) \\
\mathit{fv}(\textbf{\textit{bind}}\ x\ =\ t_1\ \textbf{\textit{in}}\ t_2) &=& \mathit{fv}(t_1) \cup \mathit{fv}(t_2) \setminus \{x\} \\
\mathit{fv}(\lambda x{:}t_1.\, t_2) &=& \mathit{fv}(t_1) \cup \mathit{fv}(t_2) \setminus \{x\} \\
\mathit{fv}(t_1\ t_2) &=& \mathit{fv}(t_1) \cup \mathit{fv}(t_2) \\
\mathit{fv}(\langle t_1, t_2\rangle) &=& \mathit{fv}(t_1) \cup \mathit{fv}(t_2)
\end{array}$$

$$\begin{array}{rcl}
\mathit{fv}(\cdot) &=& \cdot \\
\mathit{fv}(\Gamma, x : t) &=& \mathit{fv}(\Gamma) \cup \mathit{fv}(t)
\end{array}$$

A.2 Subject Reduction

Lemma A.1 (Evidence Substitution). $[\{t_2/x\}t_1] \subseteq [t_1] \cup [t_2]$

Proof. By trivial induction the structure of t_1. □

$\boxed{\Sigma \vdash \diamond}$

$$\frac{}{\cdot \vdash \diamond}\ \text{S-EMPTY} \qquad \frac{\Sigma; \cdot \vdash t : \textbf{Kind}^P}{\Sigma, a : t \vdash \diamond}\ \text{S-CONS}$$

$\boxed{\Sigma \vdash \Gamma}$

$$\frac{\Sigma \vdash \diamond}{\Sigma \vdash \cdot}\ \text{E-EMPTY}$$

$$\frac{\Sigma; \Gamma \vdash t : k \qquad \text{if } x \notin \mathit{fv}(\Gamma) \qquad k \in \{\textbf{Kind}^P, \textbf{Kind}^T, \textbf{Prop}, \textbf{Type}\}}{\Sigma \vdash \Gamma, x : t}\ \text{E-CONS}$$

Figure 6. Well formed signature and environment judgments (defined mutually with typing relation)

Lemma A.2. *If* $\vdash p \rightarrow p'$ *then* $[p'] \subseteq [p]$.

Proof. Proof by structural induction on the reduction relation.

Case R-APP1: Inverting the reduction relation gives $p = t_1\ t_2$ and $p' = t_1\ t_2$ where $\vdash t_1 \rightarrow t_1'$. By the induction hypothesis $[t_1'] \subseteq [t_1]$, so $[p'] = [t_1'] \cup [t_2] \subseteq [t_1] \cup [t_2] = [p]$.

Case R-BETA: Inverting the reduction relation gives $p = (\lambda x{:}t_1.\, t_2)\ t_3$ and $p' = \{t_3/x\}t_2$. It suffices to show $[\{t_3/x\}t_2] \subseteq [t_2] \cup [t_3]$. This follows directly from Lemma A.1.

The R-BINDT case is similar to R-BETA, and all remaining cases are similar to R-APP1. □

Lemma A.3 (Weakening). *If* $\Sigma; \Gamma, \Gamma' \vdash t_1 : t_2$ *and* $\Sigma \vdash \Gamma, x : t_3, \Gamma'$ *then* $\Sigma; \Gamma, x : t_3, \Gamma' \vdash t_1 : t_2$.

Proof. By structural induction on the typing derivation. □

Lemma A.4 (Inversion Var—Same). *If* $\Sigma; \Gamma, z : u, \Gamma' \vdash z : s$ *then* $u = s$.

Proof. Inverting the typing derivation (which ends in T-VAR) yields $z : y \in \Gamma, z : u, \Gamma'$ and $\Sigma \vdash \Gamma, z : u, \Gamma'$. Proof precedes by a trivial induction on the environment's well-formedness. □

Lemma A.5 (Inversion Var—Different). *If* $\Sigma; \Gamma, z : u, \Gamma' \vdash x : s$ *and* $u \neq z$ *then* $x : s \in \Gamma$ *or* $x : s \in \Gamma'$.

Proof. By inverting the typing derivation, than structural induction on the environment's well formedness. □

Lemma A.6 (Variables closed in context). *If* $\Sigma; \Gamma \vdash s : t$ *and* $x \in \mathit{fv}(t) \cup \mathit{fv}(s)$ *then* $x \in \mathit{dom}(\Gamma)$.

$\boxed{\vdash t \rightarrow t'}$

$$\frac{}{\vdash (\lambda x{:}t_1.\, t_2)\ t_3 \rightarrow \{t_3/x\}t_2}\ \textsc{R-beta}$$

$$\frac{x \notin \mathit{fv}(t_2)}{\vdash \textbf{bind}\ x\ =\ t_1\ \textbf{in}\ t_2 \rightarrow t_2}\ \textsc{R-bindS}$$

$$\frac{}{\vdash \textbf{bind}\ x\ =\ \textbf{return}@[t_0]\ t_1\ \textbf{in}\ t_2 \rightarrow \{t_1/x\}t_2}\ \textsc{R-bindT}$$

$$\frac{\vdash t_2 \rightarrow t_2'}{\vdash \textbf{return}@[t_1]\ t_2 \rightarrow \textbf{return}@[t_1]\ t_2}\ \textsc{R-says}$$

$$\frac{y \notin \mathit{fv}(t_3)}{\begin{array}{c}\vdash \textbf{bind}\ x\ =\ (\textbf{bind}\ y\ =\ t_1\ \textbf{in}\ t_2)\ \textbf{in}\ t_3 \rightarrow \\ \textbf{bind}\ y\ =\ t_1\ \textbf{in}\ \textbf{bind}\ x\ =\ t_2\ \textbf{in}\ t_3\end{array}}\ \textsc{R-bindC}$$

$$\frac{\vdash t_2 \rightarrow t_2'}{\vdash \lambda x{:}t_1.\, t_2 \rightarrow \lambda x{:}t_1.\, t_2'}\ \textsc{R-lam}$$

$$\frac{\vdash t_1 \rightarrow t_1'}{\vdash \textbf{bind}\ x\ =\ t_1\ \textbf{in}\ t_2 \rightarrow \textbf{bind}\ x\ =\ t_1'\ \textbf{in}\ t_2}\ \textsc{R-bind1}$$

$$\frac{\vdash t_2 \rightarrow t_2'}{\vdash \textbf{bind}\ x\ =\ t_1\ \textbf{in}\ t_2 \rightarrow \textbf{bind}\ x\ =\ t_1\ \textbf{in}\ t_2'}\ \textsc{R-bind2}$$

$$\frac{\vdash t_1 \rightarrow t_1'}{\vdash t_1\ t_2 \rightarrow t_1'\ t_2}\ \textsc{R-app1} \qquad \frac{\vdash t_2 \rightarrow t_2'}{\vdash t_1\ t_2 \rightarrow t_1\ t_2'}\ \textsc{R-app2}$$

$$\frac{\vdash t_1 \rightarrow t_1'}{\vdash \langle t_1, t_2\rangle \rightarrow \langle t_1', t_2\rangle}\ \textsc{R-pair1}$$

$$\frac{\vdash t_2 \rightarrow t_2'}{\vdash \langle t_1, t_2\rangle \rightarrow \langle t_1, t_2'\rangle}\ \textsc{R-pair2}$$

Figure 7. Reduction relation

Proof. Proof by induction on the typing derivation. □

Lemma A.7. *If* $\Sigma \vdash \Gamma, z : u$ *and* $x : s \in \Gamma$ *then* $z \notin \mathit{fv}(s)$.

Proof. By the definition of $\in$, we know $\Gamma = \Gamma_1, x : s, \Gamma_2$. The well-formedness of $\Gamma_1, x : s, \Gamma_2, z : u$ shows that (1) *dom*$(z) \notin \Gamma_1$ and (2) $\Sigma; \Gamma_1 \vdash s : k$ for some k. From these and Lemma A.6 we can conclude $z \notin \mathit{fv}(s)$. □

Lemma A.8 (Subsitution, strong form for induction)**.** *Assume* $\Sigma; \Gamma \vdash t_u : u$. *Then*

(1) $\Sigma; \Gamma, z : u, \Gamma' \vdash t : s$ *implies* $\Sigma; \Gamma, \{t_u/z\}\Gamma' \vdash \{t_u/z\}t : \{t_u/z\}s$. *And*

(2) $\Sigma \vdash \Gamma, z : u, \Gamma'$ *implies* $\Sigma \vdash \Gamma, \{t_u/z\}\Gamma'$.

Proof. By mutual structural induction over the typing and well-formed environment derivations. Proceed with inversion on the form of the last typing or well-formedness rule.

Case T-Prop. We have $t =$ **Prop** and $s =$ $\textbf{Kind}^P$. So it suffices to show $\Sigma \vdash \Gamma, \{t_u/z\}\Gamma'$. This follows immediately from the induction hypothesis.

Case T-Var. Suppose $t = z$. Then, by Lemma A.4 $s = u$. Therefore it suffices to show $\Sigma; \Gamma, \{t_u/z\}\Gamma' \vdash t_u : u$, which we get by applying weakening (finitely many times) to the assumption $\Sigma; \Gamma \vdash t_u : u$. Instead suppose $t = x \neq z$. Then we must show $\Sigma; \Gamma, \{t_u/z\}\Gamma' \vdash x : \{t_u/z\}s$. By Lemma A.5, either $x : s \in \Gamma$ or $x : s \in \Gamma'$. Suppose $x : s \in \Gamma$. Then by Lemma A.7 we find $z \notin \mathit{fv}(s)$ so $s = \{t_u/z\}s$. Thus it suffices to show $\Sigma; \Gamma, \{t_u/z\}\Gamma' \vdash x : s$, which follows from T-var, the induction hypothesis and the form of Γ. Lastly, consider the case that $x : s \in \Gamma'$. Then $x : \{t_u/z\}s \in \{t_u/z\}\Gamma'$, and we conclude using this, T-Var, and the induction hypothesis.

Case T-lam. We have $t = \lambda x{:}t_1.\, t_2$ and $s = (x{:}t_1) \rightarrow P$. Without loss of generality assume $x \neq z$. The induction hypothesis yields $\Sigma; \Gamma, \{t_u/z\}(\Gamma', x : t_1) \vdash \{t_u/z\}t_2 : \{t_u/z\}P$ and $\Sigma; \Gamma, \{t_u/z\}\Gamma' \vdash \{t_u/z\}(x{:}t_1) \rightarrow P :$ **Prop**. We conclude by applying T-lam and the following facts about substitution: $\{t_u/z\}(\Gamma', x : t_1) = (\{t_u/z\}\Gamma'), x : (\{t_u/z\}t_1)$ and $\{t_u/z\}((x{:}t_1) \rightarrow t_2) = (x{:}\{t_u/z\}t_1) \rightarrow \{t_u/z\}t_2$. (The latter holds because $x \neq z$.)

Case T-bind. This case is similar to T-lam, but uses the additional fact that, for all t_1 and t_2, $\{t_u/z\}(t_1\ \textbf{says}\ t_2) = (\{t_u/z\}t_1)\ \textbf{says}\ (\{t_u/z\}t_2)$.

Case T-sign. We have $t = \textbf{sign}(t_1, t_2)$ and $s = t_1\ \textbf{says}\ t_2$. From Lemma A.6, we find $\mathit{fv}(t_1) = \mathit{fv}(t_2) = \emptyset$. Hence $\{t_u/z\}t = t$ and $\{t_u/z\}s = s$, so to use T-sign, we need only show $\Sigma \vdash \Gamma, \{t_u/z\}\Gamma'$. This follows immediately from the induction hypothesis.

Case T-pair. We have $s = \{s_1{:}x; s_2\}$. Assume without loss of generality $x \neq z$. This case follows from the induction hypothesis and the fact $\{t_u/z\}(\{t_1/x\}s_2) = \{(\{t_u/z\}t_1)/x\}(\{t_u/z\}t_2)$.

The remaining cases are similar to T-Prop (T-Type, T-string, T-const, T-prin, T-litStr, T-litPrin), or T-lam (T-pi, T-sigma), or are trivial (T-says, T-return, T-App). □

Subject reduction will need both subsitution (above) and the following strengthening lemma (below). Note that strengthening is *not* a special case of of subsitution, as strenthening works even when u is unihabited.

Lemma A.9 (Strengthening)**.** *If* $\Sigma; \Gamma, z : u, \Gamma' \vdash t : s$ *and* $\Sigma \vdash \Gamma, \Gamma'$ *and* $z \notin \mathit{fv}(t) \cup \mathit{fv}(s)$ *then* $\Sigma; \Gamma, \Gamma' \vdash t : s$.

Proof. Proof by structural induction on the typing relation. If the typing derivation ends with T-VAR then t is a variable, x. By the defintion of $fv(\cdot)$, $x \neq z$. Inverting the typing relation yields $x : s \in \Gamma, z : u, \Gamma'$. Thus $z \in \Gamma, \Gamma'$, and we conclude with T-VAR. All other cases follow directly from the induction hypothesis. □

Lemma A.10 (Subject Reduction)**.** *If* $\vdash t \to t'$ *and* $\Sigma; \Gamma \vdash t : s$ *then* $\Sigma; \Gamma \vdash t' : s$.

Proof. Proof is by structural induction on the reduction relation. Proceed by case analysis on the last rule used.

Case R-BETA. We have $t = (\lambda x{:}t_1.\, t_2)\ t_3$ and $t' = \{t_3/x\}t_2$. Term t could only have been typed by a derivation ending in

$$\textsc{T-APP}\ \dfrac{\textsc{T-LAM}\ \dfrac{\dfrac{\vdots}{\Sigma; \Gamma, x : t_1 \vdash t_2 : s_2} \quad \ldots}{\Sigma; \Gamma \vdash \lambda x{:}t_1.\, t_2 : (x{:}t_1) \to\ s_2} \qquad \dfrac{\vdots}{\Sigma; \Gamma \vdash t_3 : s_3}}{\Sigma; \Gamma \vdash (\lambda x{:}t_1.\, t_2)\ t_3 : \{s_3/x\}s_2}$$

for some s_2 and s_3. So $s = \{t_3/x\}t_2$. That $\Sigma; \Gamma \vdash t' : s$ holds follows directly from Lemma A.8 and the judgments written in the above derivation.

Case R-BINDS. We have $t = \textbf{bind}\ x\ =\ t_1\ \textbf{in}\ t_2$ and $t' = t_2$. Term t could only be typed by T-BIND, and inverting this rule gives $s = a\ \textbf{says}\ s_2$ and $\Sigma; \Gamma, x : s_1 \vdash t_2 : a\ \textbf{says}\ s_2$. Before concluding with Lemma A.9, we must show $x \notin a\ \textbf{says}\ s_2$. This is a consequence of Lemma A.6, and the hypothesis that $a\ \textbf{says}\ s_2$ is a type assigment in Γ.

Case R-BINDT. We have $t = \textbf{bind}\ x = \textbf{return}@[t_0]\, t_1\ \textbf{in}\ t_2$ and $t' = \{t_1/x\}t_2$ and $s = t_0\ \textbf{says}\ s_2$. Term t can only be typed by a derivation ending with, for some s_1,

$$\textsc{T-BIND}\ \dfrac{\textsc{T-APP}\ \dfrac{\vdots \qquad \dfrac{\vdots}{\Sigma; \Gamma \vdash t_1 : s_1}}{\Sigma; \Gamma \vdash \textbf{return}@[t_0]\ t_1 : t_0\ \textbf{says}\ s_1} \qquad \dfrac{\vdots}{\Sigma; \Gamma, x : s_1 \vdash t_2 : t_0\ \textbf{says}\ s_2}}{\Sigma; \Gamma \vdash \textbf{bind}\ x\ =\ \textbf{return}@[t_0]\ t_1\ \textbf{in}\ t_2 : t_0\ \textbf{says}\ s_2}$$

By Lemma A.8, we find $\Sigma; \Gamma \vdash \{t_1/x\}t_2 : \{t_1/x\}(t_0\ \textbf{says}\ s_2)$. The contrapositive of Lemma A.6 shows $x \notin t_0\ \textbf{says}\ s_2$, so we can rewrite the above to $\Sigma; \Gamma \vdash t' : s$.

The remaining cases follow directly from the induction hypothesis. □

A.3 Proof of Strong Normalization

We prove Aura$_0$ is strongly normalizing by translating Aura$_0$ to the Calculus of Construction extended with product dependent types (CC).

The main property of the translation, which we will prove later in this section, is that the translation has to perserves both the typing relation and the reduction relation. The translation has the form: $[\![t]\!]_\Delta = (s, \Delta')$, where context Δ is a typing context for variables. To translate a Aura$_0$ term, we take in a context Δ, and produce a new context Δ' together with a term in CC.

The translation of Aura$_0$ terms to CC terms are defined belows. The translation collapses $\textbf{Kind}^P$ and $\textbf{Kind}^T$ to the kind $\Box$ in CC, and **Prop**, **Type** to $*$. We translate all the base types to **unit**, and constants to $eunit$. The interesting cases are the translation of DCC terms. The translation drops the monads, and tranlate the **bind** expression to lamba application. The term $\textbf{sign}(t_1, t_2)$ has type $t_1\ \textbf{says}\ t_2$; therefore, it has to be translated to a term which is the translation of t_2. One way to find such a term is to generate a fresh variable and assign its type to be the translation of t_2. The context Δ is used to keep track of the type mapping of those fresh variables generated. There are two cases in translation $\textbf{sign}(t_1, t_2)$. In the first case, the variable we need has already been generated. In the second case, we need to generate a fresh variable and append its type binding to Δ as the output context. The contexts in the translation of other terms are threaded through the translation and keep track of all the variables generated so far.

$$\boxed{[\![t]\!]_\Delta = (s, \Delta')}$$

$$\dfrac{\text{if } t \in \{\textbf{Kind}^P, \textbf{Kind}^T\}}{[\![t]\!]_\Delta = (\Box, \Delta)} \qquad \dfrac{\text{if } t \in \{\textbf{Prop}, \textbf{Type}\}}{[\![t]\!]_\Delta = (*, \Delta)}$$

$$\dfrac{\text{if } t \in \{\texttt{"a"}, \ldots, \texttt{A}\ldots\}}{[\![t]\!]_\Delta = ((\,), \Delta)} \qquad \dfrac{\text{if } t \in \{\textbf{string}, \textbf{prin}\}}{[\![t]\!]_\Delta = (\textbf{unit}, \Delta)}$$

$$[\![a]\!]_\Delta = (a, \Delta) \qquad [\![x]\!]_\Delta = (x, \Delta)$$

$$\dfrac{}{[\![t_1\ \textbf{says}\ t_2]\!]_\Delta = [\![t_2]\!]_\Delta}$$

$$\frac{[\![t_1]\!]_\Delta = (s_1, \Delta_1) \qquad [\![t_2]\!]_{\Delta_1} = (s_2, \Delta_2)}{[\![(x{:}t_1) \rightarrow\ t_2]\!]_\Delta = ((x{:}s_1) \rightarrow\ s_2, \Delta_2)}$$

$$\frac{[\![t_1]\!]_\Delta = (s_1, \Delta_1) \qquad [\![t_2]\!]_{\Delta_1} = (s_2, \Delta_2)}{[\![(x : t_1)\ \Rightarrow\ t_2]\!]_\Delta = ((x{:}s_1) \rightarrow\ s_2, \Delta_2)}$$

$$\frac{[\![t_1]\!]_\Delta = (s_1, \Delta_1) \qquad [\![t_2]\!]_{\Delta_1} = (s_2, \Delta_2)}{[\![\{x{:}t_1; t_2\}]\!]_\Delta = (\{x{:}s_1; s_2\}, \Delta_2)}$$

$$\frac{[\![t_2]\!]_\Delta = (s, \Delta_1) \qquad \Delta_1(y) = s}{[\![\textbf{sign}(t_1, t_2)]\!]_\Delta = (y, \Delta_1)}$$

$$\frac{\begin{array}{c}[\![t_2]\!]_\Delta = (s, \Delta_1)\\ \text{not exists } x \in \mathit{dom}(\Delta_1) s.t. \Delta_1(x) = s \qquad y \text{ is fresh}\end{array}}{[\![\textbf{sign}(t_1, t_2)]\!]_\Delta = (y, (\Delta_1, y : s))}$$

$$\frac{}{[\![\textbf{return}@[t_1]\ t_2]\!]_\Delta = [\![t_2]\!]_\Delta}$$

$$\frac{\begin{array}{c}[\![t_0]\!]_\Delta = (s, \Delta_1)\\ [\![t_1]\!]_{\Delta_1} = (s_1, \Delta_2) \qquad [\![t_2]\!]_{\Delta_2} = (s_2, \Delta_3)\end{array}}{[\![\textbf{bind}\ x{:}t_0\ =\ t_1\ \textbf{in}\ t_2]\!]_\Delta = ((\lambda x{:}s.\, s_2)\ s_1, \Delta_3)}$$

$$\frac{[\![t_1]\!]_\Delta = (s_1, \Delta_1) \qquad [\![t_2]\!]_{\Delta_1} = (s_2, \Delta_2)}{[\![\lambda x{:}t_1.\, t_2]\!]_\Delta = (\lambda x{:}s_1.\, s_2, \Delta_2)}$$

$$\frac{[\![t_1]\!]_\Delta = (s_1, \Delta_1) \qquad [\![t_2]\!]_{\Delta_1} = (s_2, \Delta_2)}{[\![t_1\ t_2]\!]_\Delta = (s_1\ s_2, \Delta_2)}$$

$$\frac{[\![t_1]\!]_\Delta = (s_1, \Delta_1) \qquad [\![t_2]\!]_{\Delta_1} = (s_2, \Delta_2)}{[\![\langle t_1, t_2\rangle]\!]_\Delta = (\langle s_1, s_2\rangle, \Delta_2)}$$

$\boxed{[\![\Sigma]\!]_\Delta = (\Sigma', \Delta')}$

$$\frac{}{[\![\cdot]\!]_\Delta = (\cdot, \Delta)}$$

$$\frac{[\![\Sigma]\!]_\Delta = (\Sigma', \Delta_1) \qquad [\![t]\!]_{\Delta_1} = (s, (\Delta_1, \Delta_2))}{[\![\Sigma, v : t]\!]_\Delta = ((\Sigma', \Delta_2, v : s), (\Delta_1, \Delta_2))}$$

$\boxed{[\![\Gamma]\!]_\Delta = (\Gamma', \Delta_0, \Delta')}$

$$\frac{}{[\![\cdot]\!]_\Delta = (\cdot, \cdot, \Delta)}$$

$$\frac{[\![\Gamma]\!]_\Delta = (\Gamma', \Delta_0, \Delta_1) \qquad [\![t]\!]_{\Delta_1,\Delta_2} = (s, (\Delta_1, \Delta_2, \Delta_3))}{[\![\Gamma, v : t]\!]_\Delta = ((\Gamma', v : s), (\Delta_0, \Delta_2, \Delta_3), (\Delta_1, \Delta_2, \Delta_3))}$$

Definitions

- *unique*(Δ) if for all $x, y \in \mathit{dom}(\Delta)$, $\Delta(x) \neq \Delta(y)$.
- *wf*(Γ):

$$\frac{}{\mathit{wf}(\cdot)}$$

$$\frac{\Gamma \vdash^{CC} t : s \qquad s \in \{*, \Box\} \qquad v \notin \mathit{dom}(\Gamma)}{\mathit{wf}(\Gamma, v : t)}$$

Lemma A.11 (Translation Weakening)**.** *If* $[\![t]\!]_{\Delta_1} = (s, \Delta_2)$*, unique*$(\Delta)$*, and* $\Delta_2 \subseteq \Delta$*, then* $[\![t]\!]_\Delta = (s, \Delta)$*.*

Proof. By induction on the structure of t. The key is when t is $\textbf{sign}(t_1, t_2)$.

case: $t = \textbf{sign}(t_1, t_2)$.
By assumptions,
unique(Δ) (1)
$[\![\textbf{sign}(t_1, t_2)]\!]_{\Delta_1} = (x, \Delta_2)$
and $[\![t_2]\!]_{\Delta_1} = (s, \Delta_2), \Delta_1(x) = s$ (2)
$\Delta_2 \subseteq \Delta$ (3)
By I.H. on t_1,
$[\![t_2]\!]_\Delta = (s, \Delta)$ (4)
By (2), (1), (3),
$\Delta(x) = s$ (5)
By the rules for translation,
$[\![\textbf{sign}(t_1, t_2)]\!]_\Delta = (x, \Delta)$ (6)

case: $t = \textbf{sign}(t_1, t_2)$.
By assumptions,
unique(Δ) (1)
$[\![\textbf{sign}(t_1, t_2)]\!]_{\Delta_1} = (x, (\Delta_2, x : s))$
and $[\![t_2]\!]_{\Delta_1} = (s, \Delta_2)$,
$\nexists x \in \mathit{dom}(\Delta)$ s.t. $\Delta_1(x) = s$ (2)
$(\Delta_2, x : s) \subseteq \Delta$ (3)
By I.H. on t_1,
$[\![t_2]\!]_\Delta = (s, \Delta)$ (4)
By (1), (3),
$\Delta(x) = s$ (5)
By the rules for translation,
$[\![\textbf{sign}(t_1, t_2)]\!]_\Delta = (x, \Delta)$ (6)

□

Lemma A.12 (CC Typing Weakening)**.** *If* $\Gamma_1, \Gamma_2 \vdash^{CC} t : s$*, and wf*$(\Gamma_1, \Gamma', \Gamma_2)$*, then* $\Gamma_1, \Gamma', \Gamma_2 \vdash^{CC} t : s$*.*

Proof. By induction on structure of the derivation $\mathcal{E}$ :: $\Gamma_1, \Gamma_2 \vdash^{CC} t : s$. □

Lemma A.13 (Substitution)**.** *If* $\Sigma; \Gamma \vdash t_1 : k$*,* $[\![t_1]\!]_\Delta = (s_1, \Delta)$ *and* $[\![t_2]\!]_\Delta = (s_2, \Delta)$*, then* $[\![\{t_2/x\}t_1]\!]_\Delta = (\{s_2/x\}s_1, \Delta)$*.*

Proof. By induction on the struction of t_1.

case: $t_1 = \textbf{sign}(t, p)$.
By assumption,
$[\![t_2]\!]_\Delta = (s_2, \Delta)$ (1)
$\Sigma; \Gamma \vdash \textbf{sign}(t, p) : k$ (2)
$[\![\textbf{sign}(t, p)]\!]_\Delta = (y, \Delta)$ (3)
By the definition of translation, (3),
$[\![p]\!]_\Delta = (s, \Delta)$ (4)
and $\Delta(y) = s$ (5)
By inversion on (2),
$\Sigma; \cdot \vdash p : \textbf{Prop}$ (6)
$x \notin fv(p)$ (7)
$\{t_2/x\}p = p$ (8)
By (8), (4), (5),
$[\![\{t_2/x\}(\textbf{sign}(t, p))]\!]_\Delta = (y, \Delta) = (\{s_2/x\}y, \Delta)$ (9)

$\square$

Lemma A.14 (Correctness of Translation)**.**

1. *If* $\Sigma \vdash \diamond$, $[\![\Sigma]\!]_\emptyset = (\Sigma_1, \Delta_1)$, *then* $wf(\Sigma_1)$.
2. *If* $\Sigma \vdash \Gamma$, $[\![\Sigma]\!]_\emptyset = (\Sigma_1, \Delta_1)$, $[\![\Gamma]\!]_{\Delta_1} = (\Gamma_1, \Delta_0, \Delta_2)$, *then* $wf(\Sigma_1, \Delta_0, \Gamma_1)$.
3. *If* $\mathcal{E} :: \Sigma; \Gamma \vdash t : s$, $[\![\Sigma]\!]_\emptyset = (\Sigma_1, \Delta_1)$, $[\![\Gamma]\!]_{\Delta_1} = (\Gamma_1, \Delta_0, \Delta_2)$, $[\![t]\!]_{\Delta_2,\Delta_3} = (t_1, (\Delta_2, \Delta_3, \Delta_4))$, *then* $\Sigma_1, \Delta_0, \Delta_3, \Delta_4, \Gamma_1 \vdash^{CC} t_1 : s_1$, *and* $[\![s]\!]_{(\Delta_2,\Delta_3,\Delta_4)} = (s_1, (\Delta_2, \Delta_3, \Delta_4))$.
4. *If* $\mathcal{E} :: \Sigma; \Gamma \vdash t$, $[\![\Sigma]\!]_\emptyset = (\Sigma_1, \Delta_1)$, $[\![\Gamma]\!]_{\Delta_1} = (\Gamma_1, \Delta_0, \Delta_2)$, $[\![t]\!]_{\Delta_2,\Delta_3} = (t_1, (\Delta_2, \Delta_3, \Delta_4))$, *then* $\Sigma_1, \Delta_0, \Delta_3, \Delta_4, \Gamma_1 \vdash^{CC} t_1 : */\square$.

Proof. In the proof of 1 and 2, we use 3 only when (Σ, Γ) is smaller.
1. By induction on the structure of (Σ).

case: $\Sigma = \Sigma', a : t$
By assumption,
$\Sigma', a : t \vdash \diamond$ (1)
$\Sigma; \cdot \vdash t : \textbf{Kind}^P$ (2)
$[\![\Sigma]\!]_\emptyset = (\Sigma_1, \Delta_1)$ (3)
$[\![t]\!]_{\Delta_1} = (s, (\Delta_1, \Delta_2))$ (4)
$[\![\Sigma, a : t]\!]_\emptyset = ((\Sigma_1, \Delta_2, a : s), (\Delta_1, \Delta_2))$ (5)
By 2, (2),
$\Sigma_1, \Delta_2 \vdash^{CC} s : \square$ (6)
By definition of *wf*, and (6),
$wf(\Sigma_1, \Delta_2, a : s)$ (7)

2. By induction on the structure of $(\Sigma; \Gamma)$.

case: $\Gamma = \Gamma', x : t$
By assumption,
$\Sigma \vdash \diamond$ (1)
$\Sigma \vdash \Gamma', x : t$ (2)
$[\![\Sigma]\!]_\emptyset = (\Sigma_1, \Delta_1)$ (3)
$[\![\Gamma']\!]_{\Delta_1} = (\Gamma_1, \Delta_0, \Delta_2)$ (4)
$[\![t]\!]_{\Delta_2,\Delta_3} = (s, (\Delta_2, \Delta_3, \Delta_4))$ (5)
$[\![\Sigma, \Gamma', x : t]\!]_\emptyset = ((\Sigma_1, \Gamma_1, x : s),$
$(\Delta_0, \Delta_3, \Delta_4), (\Delta_2, \Delta_3, \Delta_4))$ (6)
By inversion of (2),
$\Sigma; \Gamma' \vdash t : \textbf{Kind}^P/\textbf{Kind}^T/\textbf{Prop}/\textbf{Type}$ (7)
By 2,
$\Sigma_1, \Delta_0, \Delta_3, \Delta_4, \Gamma_1 \vdash^{CC} s : \square/*$ (8)
By definition of *wf*, and (8),
$wf(\Sigma_1, \Delta_0, \Delta_3, \Delta_4, \Gamma_1, x : s)$ (9)

3. By induction on the structure of the derivation $\mathcal{E}$.

case: $\mathcal{E}$ ends in T-PROP.
By assumption,
$$\mathcal{E} = \frac{\mathcal{E}' :: \Sigma \vdash \Gamma}{\Sigma; \Gamma \vdash \textbf{Prop} : \textbf{Kind}^P} \quad (1)$$
$[\![\Sigma]\!]_\emptyset = (\Sigma_1, \Delta_1)$ (2)
$[\![\Gamma]\!]_{\Delta_1} = (\Gamma_1, \Delta_0, \Delta_2)$ (3)
$[\![\textbf{Prop}]\!]_{\Delta_2,\Delta_3} = (*, (\Delta_2, \Delta_3))$ (4)
By *ax* rule,
$\cdot \vdash^{CC} * : \square$ (5)
By 1, (4), (2),
$wf(\Sigma_1, \Delta_0, \Delta_3, \Gamma_1)$ (6)
By Lemma weakening,
$\Sigma_1, \Delta_0, \Delta_3, \Gamma_1 \vdash^{CC} * : \square$ (7)

case: $\mathcal{E}$ ends in T-PI.
By assumption,
$$\mathcal{E} = \frac{\mathcal{E}_1 :: \Sigma; \Gamma \vdash t_1 : (\textbf{Kind}^P, \textbf{Type}, \textbf{Prop}) \qquad \mathcal{E}_2 :: \Sigma; \Gamma, x : t_1 \vdash t_2 : k_2 \quad k_2 \in \{\textbf{Kind}^P, \textbf{Prop}\}}{\Sigma; \Gamma \vdash (x{:}t_1) \rightarrow\ t_2 : k_2} \quad (1)$$
$[\![\Sigma]\!]_\emptyset = (\Sigma_1, \Delta_1)$ (2)
$[\![\Gamma]\!]_{\Delta_1} = (\Gamma_1, \Delta_0, \Delta_2)$ (3)
$[\![(x{:}t_1) \rightarrow\ t_2]\!]_{\Delta_2,\Delta_3} =$
$((x{:}s_1) \rightarrow\ s_2, (\Delta_2, \Delta_3, \Delta_4, \Delta_5))$ (4)
where $[\![t_1]\!]_{\Delta_2,\Delta_3} = (s_1, (\Delta_2, \Delta_3, \Delta_4))$ (5)
and $[\![t_2]\!]_{\Delta_2,\Delta_3,\Delta_4} = (s_2, (\Delta_2, \Delta_3, \Delta_4, \Delta_5))$ (6)
By I.H. on $\mathcal{E}_1$,
$\Sigma_1, \Delta_0, \Delta_3, \Delta_4, \Gamma_1 \vdash^{CC} s_1 : (*, \square)$ (7)
$[\![\Gamma, x : t_1]\!]_{\Delta_1} = ((\Gamma_1, x : s_1),$
$(\Delta_0, \Delta_3, \Delta_4), (\Delta_2, \Delta_3, \Delta_4))$ (8)
$\Sigma_1, \Delta_0, \Delta_2, \Delta_3, \Delta_5, \Gamma_1, x : s_1 \vdash^{CC} s_2 : (*/\square)$ (9)
By Π, (7), (9),
$\Gamma_1, \Delta_0, \Delta_2, \Delta_3, \Delta_4, \Delta_5 \vdash^{CC} (x{:}s_1) \rightarrow\ s_2 : (*/\square)$ (10)

4. By induction on the structure of the derivation $\mathcal{E}$. $\square$

The following β' reduction rule mirrors the commute reduction rule in Aura$_0$.

Special Reduction Rule:
$(\lambda x{:}t.\, t_1)((\lambda y{:}s.\, t_2)u) \rightarrow_{\beta'} (\lambda y{:}s.\, ((\lambda x{:}t.\, t_1)t2))u$

Calculus of Construction extended with product dependent types is know to be strongly normalizing [26]. We use $\mathbf{SN}(\beta)$ to denote the set of terms that are strongly normalizing under β reductions in CC; similarly, $\mathbf{SN}(\beta\beta')$ is the set of terms that are strongly normalizing under the β and β' reduction rules. We demonstrate that CC augmented with β' is also stronly normalizing.

Lemma A.15 (Strong normalization of $\beta\beta'$-reduction in CC)**.** *For all term $t \in \mathbf{SN}(\beta)$, $t \in \mathbf{SN}(\beta\beta')$.*

Proof. Use the proof technique in [10]. □

Now we prove that the reductions in CC augmented with the β' reduction rule simulates the reduction in Aura$_0$.

Lemma A.16 (Simulation)**.** *If $t \rightarrow t'$, and and $[\![t]\!]_\Delta = (s, \Delta)$, $[\![t']\!]_\Delta = (s', \Delta)$, then $s \rightarrow^+_{\beta,\beta'} s'$.*

Proof. By examing all the reduction rules. □

Lemma A.17 (Strong normalization)**.** *Aura$_0$ is strongly normalizing.*

Proof. By Lemma A.16, and Lemma A.15. A diverging path in Aura$_0$ implies a diverging path in CC. Since CC is strongly normalizing, Aura$_0$ is also strongly normalizing. □

Lemma A.18 (Weak Confluence)**.** *If $t \rightarrow t_1$, $t \rightarrow t_2$, then exists t_3 such that $t_1 \rightarrow^* t_3$, and $t_2 \rightarrow^* t_3$.*

Proof. By examing the reduction rules. □