April 2008

Evidence-Based Audit, Technical Appendix

Jeffrey A. Vaughan
University of Pennsylvania, vaughan2@seas.upenn.edu

Limin Jia
University of Pennsylvania, liminjia@seas.upenn.edu

Karl Mazurak
University of Pennsylvania, mazurak@cis.upenn.edu

Stephan A. Zdancewic
University of Pennsylvania, stevez@cis.upenn.edu

Recommended Citation

Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Stephan A. Zdancewic, "Evidence-Based Audit, Technical Appendix", . April 2008.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-09.

Evidence-Based Audit, Technical Appendix

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 reduces the trusted computing base and enables the ability to detect flaws in complex authorization policies. Moreover, the proof structure is itself useful, because proof normalization can yield information about the relevance of policy statements. Untrusted, but well-typed, applications that access resources through an appropriate interface must obey the access control policy and create proofs useful for audit.

This paper presents AURA$_0$, an authorization logic based on a dependently-typed variant of DCC and proves the metatheoretic properties of subject-reduction and normalization. It shows the utility of proof-based auditing in a number of examples and discusses several pragmatic issues that must be addressed in this context.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-09.

Evidence-based Audit, Technical Appendix

Jeffrey A. Vaughan Limin Jia Karl Mazurak Steve Zdancewic

University of Pennsylvania
Department of Compture and Information Science
Technical Report Number MS-CIS-08-09

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 reduces the trusted computing base and enables the ability to detect flaws in complex authorization policies. Moreover, the proof structure is itself useful, because proof normalization can yield information about the relevance of policy statements. Untrusted, but well-typed, applications that access resources through an appropriate interface must obey the access control policy and create proofs useful for audit.*

This paper presents AURA$_0$, *an authorization logic based on a dependently-typed variant of DCC and proves the metatheoretic properties of subject-reduction and normalization. It shows the utility of proof-based auditing in a number of examples and discusses several pragmatic issues that must be addressed in this context.*

1 Introduction

Logging, *i.e.* recording for subsequent audit significant events that occur during a system's execution, has long been recognized as a crucial part of building secure systems. A typical use of logging is found in a firewall, which might record 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. Logs are useful because they can help administrators audit the system both to identify sources of unusual or malicious behavior and to find flaws in the authorization policies enforced by the system.

Despite the practical importance of auditing, 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 [29, 11], efficiently searching confidential logs [32], and experimental research on effective, practical logging [6, 26]. 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 [33] 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. Following an abundance of prior work on authorization logic [4, 24, 17, 1, 27, 2, 21], we adopt the stance that log entries should contain *proofs* that access should be granted. Indeed, the idea of logging such proofs is implicit in the proof-carrying authorization literature [5, 7, 10], but, to our knowledge, the use of proofs for auditing purposes has not been studied outright.

There are several compelling reasons why it is advantageous 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 size of the trusted computing base; if every access-restricting function automatically logs its arguments and re-

[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.

sult, the reasoning behind any particular grant of access cannot be obscured by a careless or malicious programmer.

The impetus for this paper stems from our experience with the (ongoing) design and implementation of a new security-oriented programming language called AURA [25]. The primary goal of this work is to find mechanisms that can be used to simplify the task of manipulating authorization proofs and to ensure that appropriate logging is always performed regardless of how a reference monitor is implemented. Among other features intended to make building secure software easier, AURA provides a built-in notion of principals, and its type system treats authorization proofs as first-class objects; the authorization policies may themselves depend on program values.

This paper 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 size of 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 [2] and similar to that found in the work by Gordon, Fournet, and Maffeis [19, 20]. 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 role in this context. Of particular note is the normalization result for AURA$_0$ authorization proofs.

Section 4 presents an extended example of a file system interface; as long as a client cannot circumvent this interface, any reference monitor code is guaranteed to provide appropriate logging information. This example also demonstrates how additional domain-specific rules can be built on top of the general kernel interface, and how the logging of proofs can be useful when it isn't obvious which of these rules are appropriate.

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 future research directions. Section 6 discusses related work. The appendix contains definitions and proofs elided from the accompanying conference paper [31].

2 Kernel Mediated Access Control

A common system design idiom protects a resource with a *reference monitor*, which takes requests from (generally) untrusted clients and decides whether to allow or deny access to the resource [12]. Ideally a reference monitor should be configured using a well-specified set of *rules* that define the current access-control policy and mirror the intent of some institutional policy.

Unfortunately, access-control decisions 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, the rules must 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.

The first and second points 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—even in simple systems—is generally undecidable [23]. The third point recognizes that reference monitors may be complex and consequently vulnerable to implementation flaws.

The AURA programming model suggests a different approach to protecting resources, 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 (made up of the op_is in the figure) wraps each of the resource's native operations (the $\mathsf{raw\text{-}op}_i$s) with new operations taking an additional argument—a proof that access is permitted. Σ_K and Σ_{ext} contain constant predicate symbols that may be occur in these proofs.

Unlike in the standard reference monitor model, an AURA kernel forwards every well-typed request to its underlying resource. Each op_i function takes as an additional argument a proof that the operation is permitted and returns a corresponding proof that the operation was performed, so the well-typedness of a call ensures that the requested access is permitted. Proofs can be typechecked dynamically

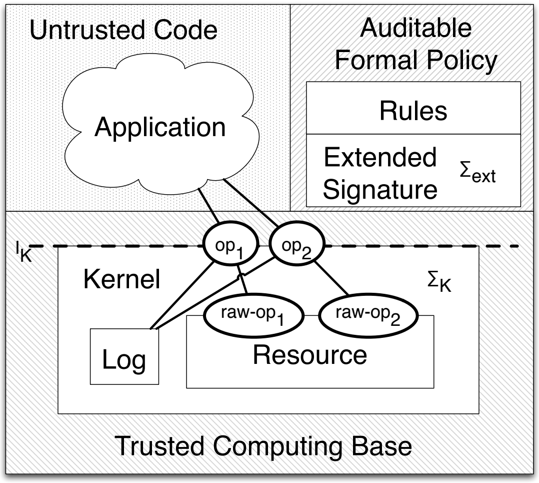

Figure 1. A monolithic application decomposed into several components operating with various degrees of trust.

in time linearly proportional to the size of the proof should the request not come from a well-typed application. Moreover, logging these proofs is enough to allow an auditor to ascertain precisely why any particular access was allowed.

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 [25], which itself is a polymorphic and dependent variant of Abadi's Dependency Core Calculus [3, 2]. In AURA$_0$, software components may be explicitly associated with one or more principals. Typically, a trusted kernel is identified with principal K, and an untrusted application may work on behalf of several principals: A, B, etc. Principals can make assertions; for instance, the (inadvisable) rule "the kernel asserts that all principals may open any file," is written as proposition K **says** $((A{:}\mathbf{prin}) \to (f{:}\mathbf{string}) \to \mathsf{OkToOpen}\ A\ f)$. Evidence for this rule contains one or more signature objects—possibly implemented as cryptographic signatures—that irrefutably tie principals to their utterances.

The above design carries several benefits. Kernels log the reasoning used to reach access control decisions; 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 than traditional, full-scale reference monitors.

A key design principle is that kernels should be 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.

2.1 The formal system description

We model a system consisting of a trusted kernel K wrapping a security-oblivious resource R and communicating with an untrusted application. The kernel is the trusted system component that mediates between the application and resource by checking and logging access control proofs; we assume that applications are prevented from accessing resources directly by using standard resource isolation techniques deployed in operating systems or type systems.

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

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

The current state σ 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. The transition function δ describes how the raw operations update state, as well as their input-output behavior. For instance, $(u, \sigma') = \delta(\mathsf{raw\text{-}op}_i, v, \sigma)$ when raw operation i—given input v and initial resource state σ—produces output u and updates the resource state to σ'.

We formalize a trusted kernel K as a tuple (L, R, Σ_K, I_K); the authority of the kernel is denoted by the constant principal 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, $\mathsf{OkToOp}_i : T_i \to \mathbf{Prop}$ and $\mathsf{DidOp}_i : T_i \to S_i \to \mathbf{Prop}$ for each $\mathsf{raw\text{-}op}_i$ of type $T_i \Rightarrow S_i$ in I_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 raw-op is permitted with input t, and a proof of K **says** $\mathsf{DidOp}\ t\ s$ means that raw-op was run with input t and returned s. Lastly, the kernel exposes an application programming interface I_K, which contains a security-aware wrapper operation

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

for each $\mathsf{raw\text{-}op}_i$ in I_R. Applications must access R through I_K rather than I_R.

The type of op_i shows that the kernel requires two arguments before it will provide access to $\mathsf{raw\text{-}op}_i$. The first argument is simply $\mathsf{raw\text{-}op}_i$'s input; the second is a proof that the kernel approves the operation, typically a composition of policy rules (globally known statements signed by

K) and statements made by other relevant principals. 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, affirming that the kernel called $\mathsf{raw\text{-}op}_i$ and linking the call's input and output. Note that OkToOp_i and DidOp_i depend on the arguments x and y.

The final components in the model are the application, the rule set, and the extended signature. We assume either that the application is well-typed—and thus that it respects I_K—or, equivalently, that the kernel performs dynamic typechecking on incoming untrusted arguments. The rule set is simply a well-known set of proofs intended to represent some access control policy; the extended signature (Σ_{ext} in Figure 1) defines predicate symbols that these rules may use in addition to those defined in Σ_K.

Remote procedure call example Consider a simple remote procedure call resource with only the single raw operation, $\mathsf{raw\text{-}rpc} : \textbf{string} \Rightarrow \textbf{string}$. The kernel associated with this resource exposes the following predicates:

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

and the kernel interface

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

A trivial policy could allow remote procedure call. This policy is most simply realized by the singleton rule set $\mathit{Rules} = \{r_0 : \mathsf{K}\ \textbf{says}\ ((x{:}\textbf{string}) \rightarrow\ \mathsf{OkToRPC}\ x)\}$.

2.2 State transition semantics

While the formalism presented thus far is sufficient to describe what AURA$_0$ systems look like at one instant in time, it is much more interesting to consider an evolving system. Here we describe variant operational semantics of the AURA$_0$ system at a semi-formal level, with emphasis on logging. The full AURA language includes a computation fragment capable of expressing the ideas in this section by way of a standard monadic state encoding, although its analysis by Jia and colleagues [25] does not address logging directly.

To demonstrate the key components of authorization and auditing in AURA$_0$, we consider evaluations from three perspectives listed as follows. In each we will consider updating states according to the transition relations defined in Figure 2.

1. Resource evaluation, written with $-\!\{\}\!\!\rightarrow^r$, models the state transition for raw resources. This relation does no logging and does not consider access control.
2. Logged evaluation, written with $-\!\{\}\!\!\rightarrow^l$, models state transitions of an AURA$_0$ system implementing logging as described in this paper. All proofs produced or consumed by the kernel are recorded in the log.
3. Semi-logged evaluation, written with $-\!\{\}\!\!\rightarrow^s$, models the full system update with weaker logging. While proofs are still required for access control, the log contains only operation names, not the associated proofs.

Resource evaluation is the simplest evaluation system. A transition $I_R; \delta \vdash \sigma -\!\{\mathsf{raw\text{-}op}_i v\}\!\!\rightarrow^r \sigma'$ may occur when v is a well typed input for $\mathsf{raw\text{-}op}_i$ according to resource interface I_R and δ specifies that $\mathsf{raw\text{-}op}_i$, given v and starting with a resource in state σ, returns u and updates the resource state to σ'. (In the following we will generally omit the $\vdash$ and objects to its left, as they are constant and can be inferred from context.)

The logged evaluation relation is more interesting: instead of simply updating resource states, it updates configurations. A configuration C, is a triple $(L, \sigma, \mathcal{S})$, where L is a list of proofs representing a log, σ is an underlying resource state, and $\mathcal{S}$ is a set of proofs of the form $\textbf{sign}(A, P)$ intended to track all assertions made by principals. There are two logged evaluation rules, L-SAY and L-ACT.

Intuitively, L-SAY allows principals other than the kernel K to add objects of the form $\textbf{sign}(A, P)$ to $\mathcal{S}$, corresponding to the ability of clients to sign arbitrary propositions, as long as all of signatures found within P already appear in $\mathcal{S}$. This last condition is written $\mathcal{S} \vDash P$ and prevents principals from forging evidence—in particular, from forging evidence signed by K. $\mathcal{S} \vDash P$ holds when all signatures embedded in P appear in $\mathcal{S}$.

Rule L-ACT models the use of a resource through it's public interface. The rules ensure that both of the operation's arguments—the data component v and the proof p—are well typed, and all accepted access control proofs are appended to the log. After the resource is called through its raw interface, the kernel signs a new proof term, q, as a receipt; it is both logged and added to $\mathcal{S}$. Again, the premise $\mathcal{S} \vDash p$ guarantees the unforgeability of **sign** objects.

The semi-logged relation functions similarly (see rules S-SAY and S-ACT), although it logs only the list of operations performed rather than any proofs.

By examining the rules in Figure 2, we can see that the kernel may only sign DidOp receipts during evaluation. Since statements signed by any other principal may be added to $\mathcal{S}$ at any time, we may identify the initial set of sign objects in $\mathcal{S}$ with the system's policy rules.

Audit and access control The three transition relations permit different operations and record different information about allowed actions. Resource evaluation allows all well-typed calls to the raw interface, and provides no information

$$\boxed{\text{Resource evaluation relation}\quad \cdot;\cdot\vdash\cdot-\!\{\cdot\}\!\!\rightarrow^{r}\cdot}$$

$$\frac{\cdot;\cdot\vdash v:T \qquad \mathsf{raw\text{-}op}_i : T \;\Rightarrow\; S \in I_R \qquad (\_,\sigma') = \delta(\mathsf{raw\text{-}op}_i, v, \sigma)}{I_R;\delta\vdash\sigma-\!\{\mathsf{raw\text{-}op}_i\ v\}\!\!\rightarrow^{r}\sigma'}\ \text{R-ACT}$$

$$\boxed{\text{Semi-logged evaluation relation}\quad \cdot;\cdot;\cdot\vdash\cdot-\!\{\cdot\}\!\!\rightarrow^{s}\cdot}$$

$$\frac{\begin{array}{c}\mathsf{op}_i : (x:T) \;\Rightarrow\; \mathsf{K}\ \textbf{says}\ \mathsf{OkToOp}_i\ x \;\Rightarrow\; \{y{:}S;\mathsf{K}\ \textbf{says}\ \mathsf{DidOp}_i\ x\ y\} \in I_{\mathsf{K}} \qquad \mathcal{S}\vDash p\\ \cdot;\cdot\vdash v:T \qquad \Sigma_{ext};\cdot\vdash p:\mathsf{K}\ \textbf{says}\ \mathsf{OkToOp}_i\ v \qquad (u,\sigma') = \delta(\mathsf{raw\text{-}op}_i, v, \sigma) \qquad q = \textbf{sign}(\mathsf{K}, \mathsf{DidOp}_i\ v\ u)\end{array}}{\Sigma_{ext};I_{\mathsf{K}};\delta\vdash(L,\sigma,\mathcal{S})-\!\{\mathsf{op}_i, v, p\}\!\!\rightarrow^{s}(\mathsf{op}_i :: L, \sigma', \mathcal{S}\cup\{q\})}\ \text{S-ACT}$$

$$\frac{\Sigma_{ext};\cdot\vdash P:\textbf{Prop} \qquad A\neq\mathsf{K} \qquad \mathcal{S}\vDash P}{\Sigma_{ext};I_{\mathsf{K}};\delta\vdash(L,\sigma,\mathcal{S})-\!\{\text{assert:}A\ \textbf{says}\ P\}\!\!\rightarrow^{s}(L,\sigma,\mathcal{S}\cup\{\textbf{sign}(A,P)\})}\ \text{S-SAY}$$

$$\boxed{\text{Proof-logged evaluation relation}\quad \cdot;\cdot;\cdot\vdash\cdot-\!\{\cdot\}\!\!\rightarrow^{l}\cdot}$$

$$\frac{\begin{array}{c}\mathsf{op}_i : (x:T) \;\Rightarrow\; \mathsf{K}\ \textbf{says}\ \mathsf{OkToOp}_i\ x \;\Rightarrow\; \{y{:}S;\mathsf{K}\ \textbf{says}\ \mathsf{DidOp}_i\ x\ y\} \in I_{\mathsf{K}} \qquad \mathcal{S}\vDash p\\ \cdot;\cdot\vdash v:T \qquad \Sigma_{ext};\cdot\vdash p:\mathsf{K}\ \textbf{says}\ \mathsf{OkToOp}_i\ v \qquad (u,\sigma') = \mathsf{raw\text{-}op}_i(v, \sigma) \qquad q = \textbf{sign}(\mathsf{K}, \mathsf{DidOp}_i\ v\ u)\end{array}}{\Sigma_{ext};I_{\mathsf{K}};\delta\vdash(L,\sigma,\mathcal{S})-\!\{\mathsf{op}_i, v, p\}\!\!\rightarrow^{l}(q :: p :: L, \sigma', \mathcal{S}\cup\{q\})}\ \text{L-ACT}$$

$$\frac{\Sigma_{ext};\cdot\vdash P:\textbf{Prop} \qquad A\neq\mathsf{K} \qquad \mathcal{S}\vDash P}{\Sigma_{ext};I_{\mathsf{K}};\delta\vdash(L,\sigma,\mathcal{S})-\!\{\text{assert:}A\ \textbf{says}\ P\}\!\!\rightarrow^{l}(L,\sigma,\mathcal{S}\cup\{\textbf{sign}(A,P)\})}\ \text{L-SAY}$$

Figure 2. Operational semantics

to auditors. Semi-logged evaluation allows only authorized access to the raw interface via access control, and provides audit information of the list of allowed operations. Logged evaluation, like semi-logged evaluation, allows only authorized access to the raw interface; it also produces a more informative log of the proofs of the authorization decisions. Intuitively, semi-logged and logged evaluation, which deploy access control, allow strictly fewer operations than resource evaluation. Logged evaluation provides more information than the semi-logged evaluation for auditing, and semi-logged evaluation provides more information than resource evaluation.

The rest of this section sketches a technical framework in which the above claims are formalized and verified. The main result, Lemma 2.1, states that logged evaluation provides more information during audit than resource evaluation; similar results hold when comparing the logged and semi-logged relations or the semi-logged and resource relations. Before we present the formal statement of this lemma, we define a few auxiliary concepts.

Each of the three relations can be lifted to define *traces*. For instance, a resource trace is a sequence of the form

$$\tau = \sigma_0-\!\{\mathsf{raw\text{-}op}_1\ v_1\}\!\!\rightarrow^{r}\sigma_1\cdots-\!\{\mathsf{raw\text{-}op}_n\ v_n\}\!\!\rightarrow^{r}\sigma_n$$

Logged and semi-logged traces are defined similarly.

The following meta-function, pronounced "erase", shows how a logged trace is implemented in terms of its encapsulated resource:

$$\begin{aligned}\lfloor(L,\sigma,\mathcal{S})\rfloor_{l/r} &= \sigma\\ \lfloor C-\!\{\text{assert: }\_\}\!\!\rightarrow^{l}\tau\rfloor_{l/r} &= \lfloor\tau\rfloor_{l/r}\\ \lfloor C-\!\{\mathsf{op}, v, \_\}\!\!\rightarrow^{l}\tau\rfloor_{l/r} &= \lfloor C\rfloor_{l/r}-\!\{(\mathsf{raw\text{-}op}, v)\}\!\!\rightarrow^{r}\lfloor\tau\rfloor_{l/r}\end{aligned}$$

For a set of traces, $\lfloor(H)\rfloor_{l/r}$ is defined as $\{\lfloor\tau\rfloor_{l/r}\mid\tau\in H\}$. Analogous functions can be defined to relate other pairs of evaluation schemes.

The $\sigma_0,\mathcal{S}_0$-*histories* of a configuration C, written $H^l(\sigma_0,\mathcal{S}_0,C)$, is defined as the set of all traces that terminate at configuration C and begin with an initial state of the form $(nil,\sigma_0,\mathcal{S}_0)$. The σ_0-histories of a resource state σ, written $H^r(\sigma_0,\sigma)$, is defined as the set of all resource traces that terminate at σ.

The following lemma makes precise the claim that logged evaluation is strictly more informative, for audit, than resource evaluation. It describes a thought experiment where an auditor looks at either a logged evaluation configuration or its erasure as a resource state. In either case the auditor can consider the histories leading up to his observation. The lemma shows that there are histories consistent with resource evaluation that are not consistent with logged evaluation. Intuitively, this means logged evaluation makes more distinctions than—and is more informative than—resource evaluation.

Lemma 2.1. *There exists a kernel K, extended signature Σ_{ext}, configuration $C = (L, \sigma, \mathcal{S})$, rule set $\mathcal{S}_0$, initial trace σ_0 and resource trace τ such that $\tau \in H^r(\sigma_0, \sigma)$, but $\tau \notin \lfloor (H^l(\sigma_0, \mathcal{S}_0, C)) \rfloor_{l/r}$.*

Proof Sketch. By construction. Let *States* $= \{up, down\}$, with initial state up. Pick a configuration C whose log contains six proofs and reflects a trace of the form $(\_, up, \_) \dashv \{\} \mapsto^l (\_, down, \_) \dashv \{\} \mapsto^l (\_, up, \_)$. Now consider trivial resource trace $\tau = up$. Observe that $\tau \in H^r(up, \lfloor C \rfloor_{l/r})$, but $\tau \notin H^l(C)$. □

Not surprisingly, it is possible to make similar distinctions between logged and semi-logged histories, as logged histories can ensure that a particular L-ACT step occurred, but this is not possible in the semi-logged case. As we will see in Section 3.3, this corresponds to the inability of the semi-logged system to distinguish between different proofs of the same proposition and thus to correctly assign blame.

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 [3, 2], Following the Curry-Howard isomorphism [16], AURA$_0$ types correspond to propositions relating to access control, and expressions correspond to proofs of these propositions. 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.

After defining the syntax and typing rules of AURA$_0$ and illustrating its use with a few simple access-control examples, this section gives the reduction rules for AURA$_0$ and discusses the importance of normalization with respect to auditing. It concludes with proofs of subject reduction, strong normalization and confluence for AURA$_0$; details may be found in the appendix.

3.1 Syntax

Figure 3 defines the syntax of AURA$_0$, which features two varieties of terms: access control proofs p, which are classified by corresponding propositions P of kind **Prop**, and conventional expressions e, which are classified by types T of the kind **Type**.[2] For ease of the subsequent presentation of the typing rules, we introduce two sorts, $\textbf{Kind}^{\textbf{P}}$ and $\textbf{Kind}^{\textbf{T}}$, which classify **Prop** and **Type** respectively. The base types are **prin**, the type of principals, and **string**; we use x to range over variables, and a to range over constants. String literals are `""`–enclosed ASCII symbols; A, B, C etc. denote literal principals, while principal variables are written A, B, C.

t, s	$::=$	$k \mid T \mid e$	Terms
k	$::=$	$\textbf{Kind}^{\textbf{P}} \mid \textbf{Kind}^{\textbf{T}}$	Sorts
	$\mid$	$\textbf{Prop} \mid \textbf{Type}$	Base kinds
T, P	$::=$	$\textbf{string} \mid \textbf{prin}$	Base types
	$\mid$	$x \mid a$	Variables and constants
	$\mid$	t **says** t	Says modality
	$\mid$	$(x{:}t) \rightarrow t$	Logical implication
	$\mid$	$(x{:}t) \Rightarrow t$	Computational arrows
	$\mid$	$\{x{:}t; t\}$	Dependent pair type
e, p	$::=$	`"a"` $\mid$ `"b"` $\mid \ldots$	String literals
	$\mid$	A $\mid$ B $\mid$ C ...	Principal literals
	$\mid$	$\textbf{sign}(A, t)$	Signature
	$\mid$	$\textbf{return}@[t]\ t$	Injection into **says**
	$\mid$	$\textbf{bind}\ x\ =\ t\ \textbf{in}\ t$	Reasoning under **says**
	$\mid$	$\lambda x{:}t.\, t \mid t\ t$	Abstraction, application
	$\mid$	$\langle t, t \rangle$	Pair

Figure 3. Syntax of AURA$_0$

In addition to the standard constructs for the functional dependent type $(x{:}t_1) \rightarrow t_2$, dependent pair type $\{x{:}t_1; t_2\}$, lambda abstraction $\lambda x{:}t_1.\, t_2$, function application $t_1\ t_2$, and pair $\langle t_1, t_2 \rangle$, AURA$_0$ includes a special computational function type $(x{:}t_1) \Rightarrow t_2$. Intuitively, $(x{:}t_1) \rightarrow t_2$ is used for logical implication and $(x{:}t_1) \Rightarrow t_2$ describes kernel interfaces; Section 3.2 discusses this further. We will sometimes write $t_1 \rightarrow t_2$, $t_1 \Rightarrow t_2$, and $\{t_1; t_2\}$ as a shorthand for $(x{:}t_1) \rightarrow t_2$, $(x{:}t_1) \Rightarrow t_2$, and $\{x{:}t_1; t_2\}$, respectively, when x does not appear free in t_2.

As in DCC, the modality **says** associates claims relating to access control with principals. The term $\textbf{return}@[A]\ p$ creates a proof of A **says** P from a proof of P, while $\textbf{bind}\ x\ =\ p_1\ \textbf{in}\ p_2$ allows a proof of A **says** P_1 to be used as a proof of P_1, but only within the scope of a proof of A **says** P_2. Finally, expressions of the form $\textbf{sign}(\mathsf{A}, P)$ represent assertions claimed without proof. Such an expression is indisputable evidence that P was asserted by A—rather than, for example, someone to whom A has delegated authority. Such signed assertions must be verifiable, binding (i.e. non-repudiable), and unforgeable; signature implementation strategies are discussed in Section 5.

[2] Our use of several syntactic categories in Figure 3 is purely for illustrative purposes.

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

$$\frac{\Sigma \vdash \Gamma}{\Sigma; \Gamma \vdash \textbf{Prop} : \textbf{Kind}^{\textbf{P}}}\ \text{T-P}\textsc{rop} \qquad \frac{\Sigma \vdash \Gamma}{\Sigma; \Gamma \vdash \textbf{Type} : \textbf{Kind}^{\textbf{T}}}\ \text{T-T}\textsc{ype} \qquad \frac{\Sigma \vdash \Gamma \qquad T \in \{\textbf{string}, \textbf{prin}\}}{\Sigma; \Gamma \vdash T : \textbf{Type}}\ \text{T-}\textsc{base}$$

$$\frac{\Sigma \vdash \Gamma \qquad x : t \in \Gamma}{\Sigma; \Gamma \vdash x : t}\ \text{T-}\textsc{var} \qquad \frac{\Sigma \vdash \Gamma \qquad a : t \in \Sigma}{\Sigma; \Gamma \vdash a : t}\ \text{T-}\textsc{const} \qquad \frac{\Sigma; \Gamma \vdash t_1 : \textbf{prin} \qquad \Sigma; \Gamma \vdash t_2 : \textbf{Prop}}{\Sigma; \Gamma \vdash t_1\ \textbf{says}\ t_2 : \textbf{Prop}}\ \text{T-}\textsc{says}$$

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

$$\frac{\Sigma; \Gamma \vdash t_1 : k_1 \qquad \Sigma; \Gamma, x : t_1 \vdash t_2 : k_2 \qquad k_1, k_2 \in \{\textbf{Type}, \textbf{Prop}\}}{\Sigma; \Gamma \vdash \{x{:}t_1; t_2\} : k_2}\ \text{T-}\textsc{pair}\text{T}\textsc{ype}$$

$$\frac{\Sigma \vdash \Gamma \qquad A \in \{\mathsf{A}, \mathsf{B}, \ldots\} \qquad \Sigma; \cdot \vdash t : \textbf{Prop}}{\Sigma; \Gamma \vdash \textbf{sign}(A, t) : A\ \textbf{says}\ t}\ \text{T-}\textsc{sign}$$

$$\frac{\Sigma \vdash \Gamma \qquad s \in \{\texttt{"a"}, \texttt{"b"}, \ldots\}}{\Sigma; \Gamma \vdash s : \textbf{string}}\ \text{T-}\textsc{lit}\text{S}\textsc{tr} \qquad \frac{\Sigma; \Gamma \vdash t_1 : \textbf{prin} \quad \Sigma; \Gamma \vdash t_2 : s_2 \quad \Sigma; \Gamma \vdash s_2 : \textbf{Prop}}{\Sigma; \Gamma \vdash \textbf{return}@[t_1]\ t_2 : t_1\ \textbf{says}\ s_2}\ \text{T-}\textsc{return}$$

$$\frac{\Sigma \vdash \Gamma \qquad A \in \{\mathsf{A}, \mathsf{B} \ldots\}}{\Sigma; \Gamma \vdash A : \textbf{prin}}\ \text{T-}\textsc{lit}\text{P}\textsc{rin} \qquad \frac{\Sigma; \Gamma \vdash e_1 : t\ \textbf{says}\ P_1 \quad \Sigma; \Gamma, x : P_1 \vdash e_2 : t\ \textbf{says}\ P_2 \quad x \notin fv(P_2)}{\Sigma; \Gamma \vdash \textbf{bind}\ x\ =\ e_1\ \textbf{in}\ e_2 : t\ \textbf{says}\ P_2}\ \text{T-}\textsc{bind}$$

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

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

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

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

Figure 4. The typing relation

3.2 Type system

AURA$_0$'s type system is defined in terms of constant signatures Σ, and variable typing contexts Γ, which associate types to global constants and local variables, respectively, and are written:

$$\Gamma ::= \cdot \mid \Gamma, x : t \qquad\qquad \Sigma ::= \cdot \mid \Sigma, a : t.$$

Typechecking consists of four judgments:

$\Sigma \vdash \diamond$	Signature Σ is well-formed
$\Sigma \vdash \Gamma$	Context Γ is well formed
$\Sigma; \Gamma \vdash t_1 : t_2$	Term t_1 has type t_2
$\Sigma; \Gamma \vdash t$	Computation type t is well-formed

The signature Σ is well-formed if Σ maps constants to types of sort $\textbf{Kind}^{\textbf{P}}$—in other words, 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 4. Most of the rules are straightforward, and we explain only a few key rules.

Rule T-SIGN states that a signed assertion created by the principal A signing a proposition P has type A **says** P; here, P can be any proposition, even false. More interesting, however, is when P contains a constant symbol defined in the signature Σ; as there is no introduction form for constants, there can be no proof of P within the logic, but the existence of signatures allows for terms of type A **says** P. 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.

The rule T-RETURN states that if we can construct a proof term p for proposition P, then the term $\textbf{return}@[A]\ p$ is a proof term for proposition A **says** P—in other words, all principals believe what can be independently verified. The T-BIND rule is a standard bind rule for monads and ensures that what principal A believes can only be used when reasoning from A's perspective.

The rule for the functional dependent type T-ARR restricts the kinds of dependencies allowed by the type system, ruling out functional dependencies on objects of kind **Type**. Note that, in the T-LAM rule, the type of the lambda abstraction must be of kind **Prop**. With such restrictions in place, it is rather straightforward to observe that these two rules allow us to express flexible access control rules while at the same time ruling out type level computations and preserving decidability of type checking.[3]

The interfaces between the application code and the kernel also requires a type description. For this reason, AURA$_0$ introduces a special computational arrow type, $(x{:}t_1) \Rightarrow t_2$. Computations cannot appear in proofs or propositions. This decouples AURA$_0$ proof reduction from effectful computation, and simplifies the interpretation of propositions. While AURA [25] demonstrates how to achieve similar results using a single arrow type and restrictions on applications, computation types simplify the exposition of AURA$_0$.

The typing rule T-PAIRTYPE for dependent pairs is standard and permits objects of kinds **Type** and **Prop** to be freely mixed; for simplicity we prohibit types and propositions themselves from appearing in a pair. Notice that AURA$_0$ features an introduction proof for pairs but no corresponding elimination form. While full AURA does, of course, feature such terms, AURA$_0$ uses dependent pairs only when associating proofs with the data on which they depend, and hence the elimination forms for pairs are unnecessary and have been elided for brevity.

3.3 Examples

The combination of dependent types and the **says** modality in AURA$_0$ can express many interesting policies. For instance, Abadi's encoding of speaks-for [2] is easily adopted:

$$A\ \mathsf{speaksfor}\ B \triangleq B\ \textbf{says}\ ((P{:}\textbf{Prop}) \to\ A\ \textbf{says}\ P \to P)$$

Adding dependency allows for more fine grained delegation. For example, we can encode partial delegation:

$$\mathsf{B}\ \textbf{says}\ ((x{:}\textbf{string}) \to\ \mathsf{A}\ \textbf{says}\ \mathsf{Good}\ x \to \mathsf{Good}\ x)$$

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

Recall also the Remote Procedure Call example from Section 2.1. While an application might use r_0 (of type K **says** $((x{:}\textbf{string}) \to\ \mathsf{OkToRPC}\ x)$) directly when building proofs, it could also construct a more convenient derived rule by using AURA$_0$'s **bind** to reason from K's perspective. For instance:

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

Rules like r_0 and its derivatives, however, are likely too trivial to admit interesting opportunities for audit; a more interesting policy states that any principal may perform a remote procedure call so long as that principal signs the input string. One encoding of this policy uses the extended context

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

and singleton rule set

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

Given this rule, an auditor might find the following proofs in the log:

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

As p_1 contains only A's signature, and as signatures are unforgeable, the auditor can conclude that A is responsible for the request—the ramifications of this depend on the real-world context of in question. Proof p_2 is more complicated; it contains signatures from both B and C. An administrator can learn several things from this proof.

We can simplify the analysis of p_2 by reducing it as discussed in the following section. Taking the normal form of p_2 (i.e., simplifying it as much as possible) yields

$$\begin{array}{l} p'_2 = \textbf{bind}\ z\ =\ r_1 \\ \qquad \textbf{in}\ \textbf{return}@[\mathsf{K}](z\ \texttt{"ab"}\ \mathsf{B}\ \textbf{sign}(\mathsf{B}, \mathsf{ReqRPC}\ \texttt{"ab"}). \end{array}$$

[3] Using two sorts, $\textbf{Kind}^{\textbf{T}}$ and $\textbf{Kind}^{\textbf{P}}$, makes it easy to state these restrictions on function types. Full AURA [25] implements a similar restriction using only a single sort; this makes some of its typing rules slightly heavier, but the two approaches appear largely equivalent.

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

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

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

$$\frac{\vdash t_2 \to t_2'}{\vdash \textbf{return}@[t_1]\ t_2 \to \textbf{return}@[t_1]\ t_2'}\ \text{R-SAYS}$$

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

Figure 5. Selected reduction rules

This term contains only B's signature, and hence B may be considered accountable for the action. This is exactly the ruling out of histories discussed in Section 2.2.

Proofs p_2 and p_2' illustrate a tension inherent to this computation model. A configuration whose log contains p_2 will be associated with fewer histories (i.e. those in which C make no assertions) than an otherwise similar configuration containing p_2'. While normalizing proofs inform policy analysis, it can also discard interesting information. To see this, consider how C's signature may be significant on an informal level. If the application is intended to pass normalized proofs to the kernel, then this is a sign that the application is malfunctioning. If principals are only supposed to sign certain statements, C's apparently spurious signature may indicate an violation of that policy, even if the signature was irrelevant to actual access control decisions.

3.4 Formal language properties

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.

Most of the reduction rules for AURA$_0$ are standard; selected rules can be seen in Figure 5, and the entire reduction relation can be found in the appendix. For **bind**, in addition to the standard congruence, beta reduction, and commute rules as found in monadic languages, we also include a special beta reduction rule R-BINDS. The R-BINDS rule eliminates bound proofs that are never mentioned in the **bind**'s body. Rule R-BINDS permits simplification of terms like **bind** $x\ =$ **sign**(A, P) **in** t, which are not subject to R-BINDT reductions. AURA$_0$ disallows reduction under **sign**, as signatures are intended to represent fixed objects realized, for example, via cryptographic means.

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. The proof proceeds by structural induction on the reduction relation and depends on several standard facts. Additionally, the R-BINDS cases requires a nonstandard lemma observing that we may remove a variable x from the typing context when x is not used elsewhere in the typing judgment. □

Proof normalization An expression is in *normal form* when it has no applicable reduction rules; as observed in Section 3.3, reducing a proof to its normal form can be quite useful for auditing. Proof normalization is most useful when the normalization process always terminates and every term has a unique normal form.

An expression t is *strongly normalizing* if application of any sequence of reduction rules to t always terminates. 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

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

Proof Sketch. We prove that AURA$_0$ is strongly normalizing by translating AURA$_0$ to the Calculus of Constructions extended with dependent pairs, which is known to be strongly normalizing [22], in a way that preserves both types and reduction steps. The interesting cases are the translations of terms relating to the **says** monad: **return** expressions are dropped, **bind** expressions are translated to to lambda application, and a term **sign**(t_1, t_2) is translated to a variable whose type is the translation of t_2. One subtle point is the tracking of dependency in the types of these newly introduced variables, which must be handled delicately. □

We have also proved that AURA$_0$ is confluent—i.e., that two series of reductions starting from the same term can always meet at some point. Let $t \to^* t'$ whenever $t = t'$ or t reduces to t' in one or more steps.

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 confluent, which follows immediately from inspection of the reduction rules. We then apply the well-known fact that strong normalization and weak confluence imply confluence. □

A direct consequence of these properties is that every AURA$_0$ term has a unique normal form; any algorithm for proof normalization will yield the same normal form for a given term. This implies that the set of relevant evidence—i.e., signatures—in a given proof term is also unique, an important property to have when assigning blame.

4 File System Example

As a more substantial example, we consider a file system in which file access is authorized using AURA$_0$ and log entries consist of authorization proofs. In a traditional file system, authorization decisions regarding file access are made when a file is opened, and thus we begin by considering only the open operation and only briefly consider additional operations. Our open is intended to provide flexible access control on top of a system featuring a corresponding raw-open and associated constants:

$$\begin{array}{ll} \mathsf{Mode} : \textbf{Type} & \mathsf{FileDes} : \textbf{Type} \\ & \\ \mathsf{RDONLY} : \mathsf{Mode} & \mathsf{WRONLY} : \mathsf{Mode} \\ \mathsf{APPEND} : \mathsf{Mode} & \mathsf{RDWR} : \mathsf{Mode} \end{array}$$

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

We can imagine that raw-open is part of the interface to an underlying file system with no notion of per-user access control or AURA$_0$ principals; it, of course, should 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. Type FileDes is left abstract; it classifies file descriptors—unforgeable capabilities used to access the contents of opened files.

Figure 6 shows the interface to open, the extended signature of available predicates, and the rules used to construct the proofs of type K **says** OkToOpen $\langle m, f\rangle$ (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

Kernel Signature Σ_K

$$\begin{aligned} \mathsf{OkToOpen} &: \{\mathsf{Mode}; \textbf{string}\} \to \textbf{Prop} \\ \mathsf{DidOpen} &: (x : \{\mathsf{Mode}; \textbf{string}\}) \to \\ &\quad\ \mathsf{FileDes} \to \textbf{Prop} \end{aligned}$$

Kernel Interface I_K

$$\begin{aligned} \mathsf{open} &: (x : \{\mathsf{Mode}; \textbf{string}\}) \Rightarrow \\ &\quad\ \mathsf{K}\ \textbf{says}\ \mathsf{OkToOpen}\ x \Rightarrow \\ &\quad\ \{h{:}\mathsf{FileDes}; \mathsf{K}\ \textbf{says}\ \mathsf{DidOpen}\ x\ h\} \end{aligned}$$

Additional Types in Extended Signature Σ_{ext}

$$\begin{aligned} \mathsf{Owns} &: \textbf{prin} \to \textbf{string} \to \textbf{Prop} \\ \mathsf{ReqOpen} &: \mathsf{Mode} \to \textbf{string} \to \textbf{Prop} \\ \mathsf{Allow} &: \textbf{prin} \to \mathsf{Mode} \to \textbf{string} \to \textbf{Prop} \end{aligned}$$

Rule Set R:

$$\mathsf{owner}f : \mathsf{K}\ \textbf{says}\ \mathsf{Owns}\ \mathsf{A}\ f$$

$$\begin{aligned} \mathsf{delegate} : \mathsf{K}\ \textbf{says}\ &((A : \textbf{prin}) \to (B : \textbf{prin}) \to \\ &(m : \mathsf{Mode}) \to (f : \textbf{string}) \to \\ &A\ \textbf{says}\ \mathsf{ReqOpen}\ m\ f \to \\ &\mathsf{K}\ \textbf{says}\ \mathsf{Owns}\ B\ f \to \\ &B\ \textbf{says}\ \mathsf{Allow}\ A\ m\ f \to \\ &\mathsf{OkToOpen}\ \langle m, f\rangle) \end{aligned}$$

$$\begin{aligned} \mathsf{owned} : \mathsf{K}\ \textbf{says}\ &((A : \textbf{prin}) \to (m : \mathsf{Mode}) \to \\ &(f : \textbf{string}) \to \\ &A\ \textbf{says}\ \mathsf{ReqOpen}\ m\ f \to \\ &\mathsf{K}\ \textbf{says}\ \mathsf{Owns}\ A\ f \to \\ &\mathsf{OkToOpen}\ \langle m, f\rangle) \end{aligned}$$

$$\begin{aligned} \mathsf{readwrite} : \mathsf{K}\ \textbf{says}\ &((A : \textbf{prin}) \to (B : \textbf{prin}) \to \\ &(f : \textbf{string}) \to \\ &B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDONLY}\ f \to \\ &B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{WRONLY}\ f \to \\ &B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDWR}\ f) \end{aligned}$$

$$\begin{aligned} \mathsf{read} : \mathsf{K}\ \textbf{says}\ &((A : \textbf{prin}) \to (B : \textbf{prin}) \to \\ &(f : \textbf{string}) \to \\ &B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDWR}\ f \to \\ &B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDONLY}\ f) \end{aligned}$$

$$\begin{aligned} \mathsf{write} : \mathsf{K}\ \textbf{says}\ &((A : \textbf{prin}) \to (B : \textbf{prin}) \to \\ &(f : \textbf{string}) \to \\ &B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDWR}\ f \to \\ &B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{WRONLY}\ f) \end{aligned}$$

$$\begin{aligned} \mathsf{append} : \mathsf{K}\ \textbf{says}\ &((A : \textbf{prin}) \to (B : \textbf{prin}) \to \\ &(f : \textbf{string}) \to \\ &B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDWR}\ f \to \\ &B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{APPEND}\ f) \end{aligned}$$

Figure 6. Types for the file system example

file system 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, however, are of great convenience. The rule owned relieves the file owner A from the need to create signatures of type A **says** Allow $A\ m\ f$ for files A owns, while readwrite allows a user who has acquired read and write permission for a file from different sources to open the file for reading and writing simultaneously. The rules read, write, and append do the reverse, allowing a user to drop from RDWR mode to RDONLY, WRONLY, or APPEND. These last four rules simply reflect semantic facts about constants of type Mode.

With the rules given in Figure 6 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. She could do both with a signature of type

$$\begin{aligned}&\mathsf{A}\ \textbf{says}\ (m:\mathsf{Mode}\rightarrow f:\textbf{string}\rightarrow \mathsf{K}\ \textbf{says}\ \mathsf{Owns}\ \mathsf{A}\ f\\&\quad\mathsf{B}\ \textbf{says}\ \mathsf{Allow}\ \mathsf{B}\ m\ f\rightarrow \mathsf{A}\ \textbf{says}\ \mathsf{Allow}\ \mathsf{B}\ m\ f).\end{aligned}$$

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.

One can easily imagine using logged proof terms—and in particular proof terms in normal form, as described in Section 3.3—to assist in assigning the blame for an unusual file access to the correct principals. For example, a single principal who carelessly delegates RDWR authority might be blamed more severely than two unrelated principals who unwittingly delegate RDONLY and WRONLY authority to someone who later makes use of readwrite. Examining the structure of proofs can once again allow an auditor to, in the terminology of Section 2.2, rule out certain histories.

We can also see how logging proofs might allow a system administrator to debug the rule set. The rules in Figure 6 might well be supplemented with, for example

$$\begin{aligned}\mathsf{surely}:\mathsf{K}\ \textbf{says}\ ((A:\textbf{prin})\rightarrow(B:\textbf{prin})\rightarrow\\(f:\textbf{string})\rightarrow\\B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDONLY}\ f\rightarrow\\B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{APPEND}\ f\rightarrow\\B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{RDWR}\ f)\end{aligned}$$

$$\begin{aligned}\mathsf{maybe}:\mathsf{K}\ \textbf{says}\ ((A:\textbf{prin})\rightarrow(B:\textbf{prin})\rightarrow\\(f:\textbf{string})\rightarrow\\B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{WRONLY}\ f\rightarrow\\B\ \textbf{says}\ \mathsf{Allow}\ A\ \mathsf{APPEND}\ f)\end{aligned}$$

Rule surely 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, 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 maybe, 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: maybe is clearly unacceptable if users desire to allow others to overwrite but not to append to files; otherwise, maybe may be seen as quite convenient, allowing, for examples, easy continuation of long write operations that were prematurely aborted. Examining the proofs in the log can help the administrator determine whether the inclusion of maybe best suits the needs of most users.

We have so far discussed only open, but there is still much AURA$_0$ has to offer a file system, 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 some analog to the subsumption rules read, write, and append—and perhaps also readwrite—along with, for pragmatic reasons, a means of preventing the kernel from logging file contents being read or written, as discussed in Section 5.

Close At first glance it seems that closing a file, like reading or writing, is an operation that requires only a valid file descriptor to ensure success, yet there is something more the type system can provide. 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 should require 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 its own right, with a protected interface to these rules and further rules concerning the circumstances under which 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 have treated signatures as abstract objects that may only be created by principals or programs with sufficient authority. This suggests two different implementation strategies.

The first approach is cryptographic: a **sign** object can be represented by a digital signature in public key cryptography. Each principal must be associated with a well known public key and in possession of its private counterpart; implementing rule T-SIGN reduces to calling a digital signature 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) either as a tuple containing the cryptographic signature along with A and P in plaintext, or as the cryptographic signature alone. 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) may not be possible. In the former case, inference is trivial, but proofs are generally large. Note that proof checking of **sign**s in either case involves validating digital signatures, a polynomial time operation.

An alternative implementation of signatures assumes that all principals trust some *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. Such indices can be small while still allowing for easy type inference, but such a scheme requires a closed system with a mutually trusted component. In a small system, the moderator 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 be quite difficult.

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 should last. Potentially mutable rules—which could be very important in a truly distributed setting—can even be represented by digital signatures in the presence of a revocation list.

The question remains, however, how best to integrate these concepts with AURA$_0$. One possible answer is to change nothing in the logic and simply allow for the possibility that any proof might be declared invalid at runtime due to an expired signature. Following this strategy requires operations to dynamically validate the timestamps in the signatures before logging, thereby making all kernel operations partial (i.e., able to fail due to expired proofs). 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* or *affine* types (see Bauer *et al.* for an authorization logic with linearity constraints [8]). Objects of a linear or affine type must be used exactly or at most once, respectively, making such types appropriate for granting 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 [18] 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." To handle time, 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 allow 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.

Proof normalization Proofs in normal form are useful for audit because they provide a particularly clear view of authorization decisions. Normalization, however, is an expensive operation—even for simply typed lambda calculus, the worst-case lower-bound on the complexity of the normalization is on the order of $\mathit{exp}(2, n)$, where $\mathit{exp}(2, 0) = 1$, $\mathit{exp}(2, n) = 2^{\mathit{exp}(2,(n-1))}$, etc., and n is the size of the term [30]. Furthermore, the size of a normalized proof can grow to $\mathit{exp}(2, n)$ as well. On the other hand, checking whether a proof is in normal form is linear to the size of the proof, and, in practice, non-malicious proof producers will likely create proofs that are simple to normalize. Consequently, where the normalization process should be carried out depends on the system in question.

A kernel operating in a highly untrusted environment might require all submitted proofs to be in normal form, shifting the computational burden to potentially malicious clients (as is commonly done to defend against denial of service attacks). By contrast, a kernel providing services to a "smart dust" network might normalize proofs itself, shifting work away from computationally impoverished nodes and onto a more robust system, again a standard design. Server side normalization might be done online as proofs come in (to amortized computation cost) or offline during audit (to avoid latency). Ultimately, the AURA programming model naturally accommodates these approaches and others; an implementation should allow programmers to select whatever normalization model is appropriate.

Authentication In Section 4 we assumed that signatures of type A **says** 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 must implement a secure channel on top of an insecure one. In practice this is done with cryptography, and one of the long-term goals of the AURA project is to elegantly integrate cryptographic methods with the type system, following the work of, for example, Fournet, *et al.* [20].

Pragmatics We are in the process of implementing AURA, in part to gain practical experience with the methodology proposed in this paper. Besides the 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 for proof-carrying code [28] 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 tool support for browsing querying the audit logs: tools should allow system administrators to issue queries against the log and analyze the evidence that is present and rules that have been used.

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

$$\begin{aligned}\mathsf{log} : (x : \mathbf{string}) \rightarrow \mathsf{K}\ \mathbf{says}\ \mathsf{OkToLog}\ x \rightarrow\\ \mathsf{K}\ \mathbf{says}\ \mathsf{DidLog}\ x\end{aligned}$$

with (hopefully permissive) rules for constructing OkToLog proofs. It might be especially useful to log failed attempts at proof construction. For example, users of the file system presented in Section 4 might repeatedly attempt to construct proofs for APPEND access given only the privileges necessary for WRONLY access, indicating that the rule maybe might be appropriate for their needs.

Conversely, some operations take arguments that should not be logged, perhaps due to security or space constraints. Section 4 mentions the possibility of logging file read and write operations, which touches on both these issues—even if it were practical to log all data read from and written to each file, many users would likely prefer that their file contents not be included in the system logs. Terms that must be excluded from the log, however, limit not just the scope of auditing but also the dependencies that may occur within propositions, as it would hardly suffice for data excised from the log to appear inside a type annotation.

6 Related Work

Earlier work on proof-carrying access control [4, 5, 14, 9, 10, 19] recognized the importance of **says** and provided a variety of interpretations for it. Garg and Pfenning [21] and, later, Abadi [2] introduced the treatment of **says** as an indexed monad. Both systems [21, 3] also enjoy the crucial noninterference property: in the absence of delegation, nothing B says can cause A to say false. AURA$_0$ builds on this prior work, especially Abadi's 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 built-in acts-for lattice (which can be encoded as described in Section 3.3) along with the protects relation (which allows additional commutation and simplification of **says** with with regards to that lattice).

Our work is closely related to Fournet, Gordon and Maffeis's research on authorization in distributed systems. [19, 20] 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, which are reflected into the type system using static theorem proving and a type constructor Ok. The inhabitants of Ok, however, 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 [13] are also related to our work. Trust management systems are meant to determine whether a set of credentials proves that the request complies with a security policy, and they use general purpose compliance checkers to verify these credentials. In PolicyMaker, proofs are programs—written in a safe language—that operate on strings; a request r is allowed when the application can combine proofs such that the result returns true on input r. While validity of AURA$_0$ propositions is tested by type checking, validity in PolicyMaker is tested by *evaluation*; this represents a fundamentally different approaches to logic. Similar to this paper, trust management systems intend for proof checking to occur in a small and application-independent 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 [9, 10]. 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 with the model discussed in this paper: in Grey, proof generating devices, like our applications, need not reside in the trusted computing base, and both systems use expressive foundational logics to define policies (Grey uses higher-order logic [15]). 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) [33], a practical system which shares several architectural elements with AURA$_0$. 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 also 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 K **says** rules to bootstrap a policy, while 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—rule violations.

Cederquist and colleagues describe a distributed system architecture with discretionary logging and no reference monitor [14]. 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.* formalize accountability such that agents are guilty until proved innocent—that is, agents use log entries to reduce the quantity of actions for which they can be held accountable. This relies on the ability of some authority to independently observe certain actions; such observations are 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 has presented 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$_0$ have demonstrated how we envision applying these ideas in practice.

Acknowledgements We thank the anonymous reviewers for their helpful comments. This work has been supported in part by NSF grants CNS-0524059, CCF-0524035, and CNS-0346939.

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: Proc. of the 11th International Conference on Functional Programming*, pages 263–273, 2006.

[3] Martín 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.

[4] Martín Abadi, Michael Burrows, Butler W. Lampson, and Gordon D. Plotkin. A calculus for access control in dis-

tributed systems. *Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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*, pages 431–445, 2005.

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

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

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

[13] 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.

[14] 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.

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

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

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

[18] Henry deYoung, Deepak Garg, and Frank Pfenning. An authorization logic with explicit time. In *Proc. of the 21st IEEE Computer Security Foundations Symposium*, 2008.

[19] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In *Proc. of the 14th European Symposium on Programming*, 2005.

[20] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization in distributed systems. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, 2007.

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

[22] 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, 1995.

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

[24] 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.

[25] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: A programming language for authorization and audit, preliminary technical results. Technical Report MS-CIS-08-10, U. Pennsylvania, 2008.

[26] 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.

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

[28] 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.

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

[30] Helmut Schwichtenberg. Normalization. Lecture Notes for Marktoberdorf Summer School, 1989.

[31] Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Steve Zdancewic. Evidence-based audit. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, Pittsburgh, PA, USA, 2008.

[32] 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*, 2004.

[33] 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

This appendix, not included in the conference version of the paper [31], provides definitions and proofs for AURA$_0$.

A.1 Definitions

The substitution and free variable functions are defined as usual.

Definition A.1 (Metafunctions over syntax)**.**

Substitution of term s for variable z in term t, $\{s/z\}t$:

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

Substitution of term s for variable z in context Γ, $\{s/z\}\Gamma$:

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

Free variables of term t, $\textit{fv}(t)$:

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

Free variables of context Γ, $\textit{fv}(\Gamma)$:

$$\begin{aligned}
\textit{fv}(\cdot) &= \cdot \\
\textit{fv}(\Gamma, x : t) &= \textit{fv}(\Gamma) \cup \textit{fv}(t)
\end{aligned}$$

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

$$\frac{}{\cdot \vdash \diamond}\ \textsc{S-empty} \qquad\qquad \frac{\Sigma \vdash \diamond \qquad \Sigma; \cdot \vdash t : \mathbf{Kind^P}}{\Sigma, a : t \vdash \diamond}\ \textsc{S-cons}$$

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

$$\frac{\Sigma \vdash \diamond}{\Sigma \vdash \cdot}\ \textsc{E-empty} \qquad \frac{\Sigma \vdash \Gamma \qquad \Sigma; \Gamma \vdash t : k \qquad \text{if } x \notin \mathit{fv}(\Gamma) \qquad k \in \{\mathbf{Kind^P}, \mathbf{Kind^T}, \mathbf{Prop}, \mathbf{Type}\}}{\Sigma \vdash \Gamma, x : t}\ \textsc{E-cons}$$

Figure 7. Well formed signature and environment judgments (defined mutually with typing relation)

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

$$\frac{}{\vdash (\lambda x{:}t_1.\, t_2)\ t_3 \rightarrow \{t_3/x\}t_2}\ \textsc{R-beta} \qquad \frac{x \notin \mathit{fv}(t_2)}{\vdash \mathbf{bind}\ x\ =\ t_1\ \mathbf{in}\ t_2 \rightarrow t_2}\ \textsc{R-bindS}$$

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

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

$$\frac{\vdash t_2 \rightarrow t_2'}{\vdash \lambda x{:}t_1.\, t_2 \rightarrow \lambda x{:}t_1.\, t_2'}\ \textsc{R-lam} \qquad \frac{\vdash t_1 \rightarrow t_1'}{\vdash \mathbf{bind}\ x\ =\ t_1\ \mathbf{in}\ t_2 \rightarrow \mathbf{bind}\ x\ =\ t_1'\ \mathbf{in}\ t_2}\ \textsc{R-bind1}$$

$$\frac{\vdash t_2 \rightarrow t_2'}{\vdash \mathbf{bind}\ x\ =\ t_1\ \mathbf{in}\ t_2 \rightarrow \mathbf{bind}\ x\ =\ t_1\ \mathbf{in}\ t_2'}\ \textsc{R-bind2} \qquad \frac{\vdash t_1 \rightarrow t_1'}{\vdash t_1\ t_2 \rightarrow t_1'\ t_2}\ \textsc{R-app1} \qquad \frac{\vdash t_2 \rightarrow t_2'}{\vdash t_1\ t_2 \rightarrow t_1\ t_2'}\ \textsc{R-app2}$$

$$\frac{\vdash t_1 \rightarrow t_1'}{\vdash \langle t_1, t_2 \rangle \rightarrow \langle t_1', t_2 \rangle}\ \textsc{R-pair1} \qquad \frac{\vdash t_2 \rightarrow t_2'}{\vdash \langle t_1, t_2 \rangle \rightarrow \langle t_1, t_2' \rangle}\ \textsc{R-pair2}$$

Figure 8. Reduction relation

A.2 Subject Reduction

Lemma A.1 (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.2 (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.3 (Inversion Var—Different). *If* $\Sigma; \Gamma, z : u, \Gamma' \vdash x : s$ *and* $x \neq z$ *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.4 (Variables closed in context). *If* $\Sigma; \Gamma \vdash s : t$ *and* $x \in \mathit{fv}(t) \cup \mathit{fv}(s)$ *then* $x \in \mathit{dom}(\Gamma)$.

Proof. Proof by induction on the typing derivation. □

Lemma A.5 (Context Ordering)**.** *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) $\mathit{dom}(z) \notin \Gamma_1$ and (2) $\Sigma; \Gamma_1 \vdash s : k$ for some k. From these and Lemma A.4 we can conclude $z \notin \mathit{fv}(s)$. □

Lemma A.6 (Well-formedness)**.** *(1) If* $\Sigma; \Gamma \vdash t : s$ *then* $s \in \{\textbf{\textit{Kind}}^{\textbf{\textit{P}}}, \textbf{\textit{Kind}}^{\textbf{\textit{T}}}\}$ *or there exists* k *such that* $\Sigma; \Gamma \vdash s : k$.

(2) If $\Sigma \vdash \Gamma$ *and* $x : s \in \Gamma$ *then there exists* k *such that* $\Sigma; \Gamma \vdash s : k$.

(3) If $\Sigma \vdash \diamond$ *and* $a : s \in \Gamma$ *then there exists* k *such that* $\Sigma; \cdot \vdash s : k$.

Proof. By mutual induction on the typing, well-formed signature, and well-formed environment judgments. □

Lemma A.7 (Substitution, 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 $t = \textbf{Prop}$ and $s = \textbf{Kind}^{\textbf{P}}$. 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.2 $s = u$. 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.3, either $x : s \in \Gamma$ or $x : s \in \Gamma'$. Suppose $x : s \in \Gamma$. Then by Lemma A.5 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 $\Sigma; \Gamma, \{t_u/z\}\Gamma' \vdash \{t_u/z\}(x{:}t_1) \rightarrow P : \textbf{Prop}$. 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.4, 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-ARR, T-PAIRTYPE), or are trivial (T-SAYS, T-RETURN, T-APP). □

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

Lemma A.8 (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.

Case T-VAR. Then t is a variable, x. By the definition of $\mathit{fv}(\cdot)$, $x \neq z$. Inverting the typing relation yields $x : s \in \Gamma, z : u, \Gamma'$. Thus $z \in \Gamma, \Gamma'$, and we conclude with T-VAR.

Case T-PAIR. We have $s = \{x{:}s_1; s_2\}$ for some x, s_1, and s_2. By A.6 and a simple case analysis, $\Sigma; \Gamma \vdash s : k$ where $k \in \{\textbf{Prop}, \textbf{Type}, \textbf{Kind}^{\textbf{P}}, \textbf{Kind}^{\textbf{T}}\}$. Thus $z \notin \mathit{fv}(k)$. Thus the case follows from the induction hypothesis and T-PAIR.

All other cases follow directly from the induction hypothesis. □

Lemma A.9 (Subject Reduction)**.** *If* $\vdash t \rightarrow t'$ *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

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

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.7 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.8, we must show $x \notin a\ \textbf{says}\ s_2$. This is a consequence of Lemma A.4, and the hypothesis that $a\ \textbf{says}\ s_2$ is a type assignment 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,

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

By Lemma A.7, we find $\Sigma;\Gamma \vdash \{t_1/x\}t_2 : \{t_1/x\}(t_0\ \textbf{says}\ s_2)$. The contrapositive of Lemma A.4 shows $x \notin t_0\ \textbf{says}\ s_2$, so we can rewrite the above to $\Sigma;\Gamma \vdash t' : s$.

Case R-BINDC. We have $t = \textbf{bind}\ x\ =\ (\textbf{bind}\ y\ =\ t_1\ \textbf{in}\ t_2)\ \textbf{in}\ t_3$ and $s = u\ \textbf{says}\ s_3$. Following the Barendregt variable convention, assume $y \notin \mathit{fv}(\Gamma) \cup \{x\}$. Inverting the typing derivation twice shows, for some s_1 and s_2, that $\Sigma;\Gamma \vdash t_1 : u\ \textbf{says}\ s_1$, $\Sigma;\Gamma, y : s_1 \vdash t_2 : u\ \textbf{says}\ s_2$, $\Sigma;\Gamma, x : s_2 \vdash t_3 : u\ \textbf{says}\ s_3$, and $x \notin \mathit{fv}(s_3)$. With Lemma A.1 we find $\Sigma;\Gamma, y : s_1, x : s_2 \vdash t_3 : u\ \textbf{says}\ s_3$. With Lemma A.4, $y \notin \mathit{fv}(s_3)$. We conclude using T-BINDC twice.

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 preserves both the typing relation and the reduction relation. The translation of terms 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.

Before we present the formal definitions of the translation, we define the following auxiliary definitions.

Definitions

- $\mathit{unique}(\Delta)$ if for all $\mathit{fvar}_1, \mathit{fvar}_2 \in \mathit{dom}(\Delta)$, $\Delta(\mathit{fvar}_1) \neq \Delta(\mathit{fvar}_2)$.
- $\mathit{wf}(\Gamma)$:

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

The translation of AURA$_0$ terms to CC terms is defined in Figure A.3. The translation collapses $\textbf{Kind}^{\textbf{P}}$ and $\textbf{Kind}^{\textbf{T}}$ to the kind $\Box$ in CC, and **Prop**, **Type** to $*$. We translate all base types to **unit**, and constants to $(\,)$. The interesting cases are the translation of DCC terms. The translation drops the monads, and translates the **bind** expression to lambda application. The term **sign**(t_1, t_2) has type t_1 **says** t_2; therefore, it has to be translated to a term whose type 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 **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_1 as the output context. To make proofs easier, we assume that the fresh variables are denoted by a *fvar*, not to be confused with the variable x.

Both of AURA$_0$'s signature Σ and context Γ are translated into CC's typing context. The translation of Σ has the form $[\![\Sigma]\!] = \Sigma'$. The translation of Γ context has the form $[\![\Gamma]\!]_\Sigma = (\Gamma', \Delta')$. The context Δ' contains all the fresh variables generated while translating the types in Γ. One subtlety of the context translation is that it has "weakening" built-in. Notice that in the translation of $\Gamma, v : t$, the translation of Γ yields Σ_1, Δ_1, but t is translated in the larger context $\Sigma_1, \Delta_1, \Delta_2$. This also means that the translation of context Γ is not unique. The judgment $[\![\Gamma]\!]_\Sigma = (\Gamma', \Delta')$ is more precisely read as (Γ', Δ') is a legitimate translation of Γ given Σ. This is good enough for our proof because we only need to show that for any well-typed AURA$_0$ term t, there is a typing derivation for the translation of t in CC.

Lemma A.10 (Translation Weakening)**.** *If* $[\![t]\!]_{\Delta_1} = (s, \Delta_2)$*, unique*$(\Delta)$*, and* $(\Delta_1, \Delta_2) \subseteq \Delta$*, then* $[\![t]\!]_\Delta = (s, \cdot)$.

Proof. By induction on the structure of t. The key is when t is **sign**(t_1, t_2).

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

case: $t =$ **sign**(t_1, t_2).
By assumptions,
unique(Δ) (1)
$[\![\mathbf{sign}(t_1, t_2)]\!]_{\Delta_1} = (\mathit{fvar}_1, (\Delta_2, \mathit{fvar}_1 : s))$
and $[\![t_2]\!]_{\Delta_1} = (s, \Delta_2)$,
$\nexists \mathit{fvar} \in \mathit{dom}(\Delta_2)$ s.t. $(\Delta, \Delta_2)(\mathit{fvar}) = s$ (2)
$(\Delta_1, \Delta_2, \mathit{fvar} : s) \subseteq \Delta$ (3)
By I.H. on t_2,
$[\![t_2]\!]_\Delta = (s, \cdot)$ (4)
By (1), (3),
$\Delta(\mathit{fvar}_1) = s$ (5)
By the rules for translation,
$[\![\mathbf{sign}(t_1, t_2)]\!]_\Delta = (\mathit{fvar}_1, \cdot)$ (6)

□

Lemma A.11 (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.12 (CC well-formed term gives well-formed environment)**.** *If* $\Gamma \vdash^{CC} t : s$ *then wf*(Γ).

Proof. By induction on the typing derivation. □

Lemma A.13 (CC well-formed term gives well-formed type)**.** *If* $\Gamma \vdash^{CC} t : s$ *then either* $s = \Box$ *or exists* k *such that* $\Gamma \vdash^{CC} s : k$.

$$\boxed{\llbracket t \rrbracket_\Delta = (s, \Delta')}$$

$$\frac{\text{if } t \in \{\mathbf{Kind^P}, \mathbf{Kind^T}\}}{\llbracket t \rrbracket_\Delta = (\Box, \cdot)} \qquad \frac{\text{if } t \in \{\mathbf{Prop}, \mathbf{Type}\}}{\llbracket t \rrbracket_\Delta = (*, \cdot)} \qquad \frac{\text{if } t \in \{\texttt{"a"}, \ldots, \texttt{A}\ldots\}}{\llbracket t \rrbracket_\Delta = ((\,), \cdot)} \qquad \frac{\text{if } t \in \{\mathbf{string}, \mathbf{prin}\}}{\llbracket t \rrbracket_\Delta = (\mathbf{unit}, \cdot)}$$

$$\llbracket a \rrbracket_\Delta = (a, \cdot) \qquad \llbracket x \rrbracket_\Delta = (x, \cdot) \qquad \frac{}{\llbracket t_1 \ \mathbf{says}\ t_2 \rrbracket_\Delta = \llbracket t_2 \rrbracket_\Delta}$$

$$\frac{\llbracket t_1 \rrbracket_\Delta = (s_1, \Delta_1) \qquad \llbracket t_2 \rrbracket_{\Delta,\Delta_1} = (s_2, \Delta_2)}{\llbracket (x{:}t_1) \rightarrow\ t_2 \rrbracket_\Delta = ((x{:}s_1) \rightarrow\ s_2, (\Delta_1, \Delta_2))} \qquad \frac{\llbracket t_1 \rrbracket_\Delta = (s_1, \Delta_1) \qquad \llbracket t_2 \rrbracket_{\Delta,\Delta_1} = (s_2, \Delta_2)}{\llbracket (x : t_1)\ \Rightarrow\ t_2 \rrbracket_\Delta = ((x{:}s_1) \rightarrow\ s_2, (\Delta_1, \Delta_2))}$$

$$\frac{\llbracket t_1 \rrbracket_\Delta = (s_1, \Delta_1) \qquad \llbracket t_2 \rrbracket_{\Delta,\Delta_1} = (s_2, \Delta_2)}{\llbracket \{x{:}t_1; t_2\} \rrbracket_\Delta = (\{x{:}s_1; s_2\}, (\Delta_1, \Delta_2))} \qquad \frac{\llbracket t_2 \rrbracket_\Delta = (s, \Delta_1) \qquad (\Delta, \Delta_1)(\mathit{fvar}) = s}{\llbracket \mathbf{sign}(t_1, t_2) \rrbracket_\Delta = \mathit{fvar}, \Delta_1)}$$

$$\frac{\llbracket t_2 \rrbracket_\Delta = (s, \Delta_1) \qquad \text{not exists}\, \mathit{fvar} \in \mathit{dom}(\Delta, \Delta_1) s.t. (\Delta, \Delta_1)(\mathit{fvar}) = s \qquad y \text{ is fresh}}{\llbracket \mathbf{sign}(t_1, t_2) \rrbracket_\Delta = (freshvar, (\Delta_1, y : s))}$$

$$\frac{}{\llbracket \mathbf{return}@[t_1]\ t_2 \rrbracket_\Delta = \llbracket t_2 \rrbracket_\Delta} \qquad \frac{\llbracket t_0 \rrbracket_\Delta = (s, \Delta_1) \qquad \llbracket t_1 \rrbracket_{\Delta,\Delta_1} = (s_1, \Delta_2) \qquad \llbracket t_2 \rrbracket_{\Delta,\Delta_1,\Delta_2} = (s_2, \Delta_3)}{\llbracket \mathbf{bind}\ x{:}t_0\ =\ t_1\ \mathbf{in}\ t_2 \rrbracket_\Delta = ((\lambda x{:}s.\, s_2)\ s_1, (\Delta_1, \Delta_2, \Delta_3))}$$

$$\frac{\llbracket t_1 \rrbracket_\Delta = (s_1, \Delta_1) \qquad \llbracket t_2 \rrbracket_{\Delta,\Delta_1} = (s_2, \Delta_2)}{\llbracket \lambda x{:}t_1.\, t_2 \rrbracket_\Delta = (\lambda x{:}s_1.\, s_2, (\Delta_1, \Delta_2))} \qquad \frac{\llbracket t_1 \rrbracket_\Delta = (s_1, \Delta_1) \qquad \llbracket t_2 \rrbracket_{\Delta,\Delta_1} = (s_2, \Delta_2)}{\llbracket t_1\ t_2 \rrbracket_\Delta = (s_1\ s_2, (\Delta_1, \Delta_2))}$$

$$\frac{\llbracket t_1 \rrbracket_\Delta = (s_1, \Delta_1) \qquad \llbracket t_2 \rrbracket_{\Delta,\Delta_1} = (s_2, \Delta_2)}{\llbracket \langle t_1, t_2 \rangle \rrbracket_\Delta = (\langle s_1, s_2 \rangle, (\Delta_1, \Delta_2))}$$

$$\boxed{\llbracket \Sigma \rrbracket = \Sigma'}$$

$$\frac{}{\llbracket \cdot \rrbracket = \cdot} \qquad \frac{\llbracket \Sigma \rrbracket = \Sigma' \qquad \llbracket t \rrbracket_\Sigma = (s, \Delta_2)}{\llbracket \Sigma, v : t \rrbracket = (\Sigma', \Delta_2, v : s)}$$

$$\boxed{\llbracket \Gamma \rrbracket_\Sigma = (\Gamma', \Delta)}$$

$$\frac{}{\llbracket \cdot \rrbracket_\Sigma = (\cdot, \cdot)} \qquad \frac{\llbracket \Gamma \rrbracket_\Sigma = (\Gamma', \Delta_1) \qquad \mathit{wf}(\Sigma, \Delta_1, \Delta_2) \qquad \mathit{unique}(\Sigma, \Delta_1, \Delta_2) \qquad \llbracket t \rrbracket_{\Sigma,\Delta_1,\Delta_2} = (s, \Delta_3)}{\llbracket \Gamma, v : t \rrbracket_\Sigma = ((\Gamma', v : s), (\Delta_1, \Delta_2, \Delta_3))}$$

Figure 9. Translation of AURA$_0$'s terms and contexts to CC

Proof. By induction on the typing derivation. □

Lemma A.14 (Substitution)**.** *If* $\Sigma; \Gamma \vdash t_1 : k$, *unique*$(\Delta)$, $\llbracket t_1 \rrbracket_\Delta = (s_1, \cdot)$ *and* $\llbracket t_2 \rrbracket_\Delta = (s_2, \cdot)$, *then* $\llbracket \{t_2/x\} t_1 \rrbracket_\Delta = (\{s_2/x\} s_1, \cdot)$.

Proof. By induction on the structure of t_1.

case: $t_1 = \mathbf{sign}(t, p)$.

By assumption,

$\llbracket t_2 \rrbracket_\Delta = (s_2, \cdot)$ (1)

$\Sigma; \Gamma \vdash \mathbf{sign}(t, p) : k$ (2)

$\llbracket \mathbf{sign}(t, p) \rrbracket_\Delta = (\mathit{fvar}, \cdot)$ (3)

By the definition of translation, (3),

$\llbracket p\rrbracket_{\Delta} = (s, \cdot)$ (4)

and $\Delta(\mathit{fvar}) = s$ (5)

By inversion on (2),

$\Sigma; \cdot \vdash p : \textbf{Prop}$ (6)

$x \notin \mathit{fv}(p)$ (7)

$\{t_2/x\}p = p$ (8)

By (8), (4), (5),

$\llbracket \{t_2/x\}(\textbf{sign}(t, p))\rrbracket_{\Delta} = (\mathit{fvar}, \cdot) = (\{s_2/x\}\mathit{fvar}, \cdot)$ (9)

□

Lemma A.15 (Correctness of Translation)**.**

1. *If* $\Sigma \vdash \diamond$, $\llbracket\Sigma\rrbracket = \Sigma_1$, *then wf*$(\Sigma_1)$ *and unique*(Σ_1).
2. *If* $\Sigma \vdash \Gamma$, $\llbracket\Sigma\rrbracket = \Sigma_1$, $\llbracket\Gamma\rrbracket_{\Sigma_1} = (\Gamma_1, \Delta)$, *then wf*$(\Sigma_1, \Delta, \Gamma_1)$ *and unique*(Σ_1, Δ).
3. *If* $\mathcal{E} :: \Sigma; \Gamma \vdash t : s$, $\llbracket\Sigma\rrbracket = \Sigma_1$, $\llbracket\Gamma\rrbracket_{\Sigma_1} = (\Gamma_1, \Delta_1)$, *wf*$(\Sigma_1, \Delta_1, \Delta_2, \Gamma_1)$, *unique*$(\Sigma_1, \Delta_1, \Delta_2)$, $\llbracket t\rrbracket_{\Sigma_1, \Delta_1, \Delta_2} = (t_1, \Delta_3)$, *then* $\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1 \vdash^{CC} t_1 : s_1$, *and* $\llbracket s\rrbracket_{(\Sigma_1, \Delta_2, \Delta_3)} = (s_1, \cdot)$ *and unique*$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)$.
4. *If* $\mathcal{E} :: \Sigma; \Gamma \vdash t$, $\llbracket\Sigma\rrbracket = \Sigma_1$, $\llbracket\Gamma\rrbracket_{\Sigma_1} = (\Gamma_1, \Delta_1)$, *wf*$(\Sigma_1, \Delta_1, \Delta_2, \Gamma_1)$, *unique*$(\Sigma_1, \Delta_1, \Delta_2)$, $\llbracket t\rrbracket_{\Sigma_1, \Delta_1} = (t_1, \Delta_3)$, *then* $\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1 \vdash^{CC} t_1 : */\Box$ *(read as* t_3 *is classified by* $*$ *or* $\Box$*), and unique*$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)$.

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

case: $\Sigma = \Sigma', a : t$

By assumption,

$\Sigma', a : t \vdash \diamond$ (1)

$\llbracket\Sigma', a : t\rrbracket = (\Sigma_1, \Delta, a : s)$ (2)

where $\llbracket\Sigma'\rrbracket = \Sigma_1$ (3)

and$\llbracket t\rrbracket_{\Sigma_1} = (s, \Delta)$ (4)

By inversion of (1),

$\Sigma' \vdash \diamond$ (5)

$\Sigma'; \cdot \vdash t : \textbf{Kind}^{\textbf{P}}$ (6)

By I.H. on Σ',

wf(Σ_1) and *unique*(Σ_1) (7)

By 3, (6), (7), (4),

$\Sigma_1, \Delta \vdash^{CC} s : \Box$ and *unique*(Σ_1, Δ) (8)

By definition of *wf* and *unique*, and (8),

wf$(\Sigma_1, \Delta, a : s)$, and *unique*$(\Sigma_1, \Delta, a : s)$ (9)

2. By induction on the structure of Γ.

case: $\Gamma = \Gamma', x : t$

By assumption,

$\Sigma \vdash \Gamma', x : t$ (1)

$\llbracket\Sigma\rrbracket = \Sigma_1$ (2)

$\llbracket\Gamma', x : t\rrbracket_{\Sigma_1} = ((\Gamma_1, x : s), (\Delta_1, \Delta_2, \Delta_3))$ (3)

where $\llbracket\Gamma'\rrbracket_{\Sigma_1} = (\Gamma_1, \Delta_1)$ (4)

and *wf*$(\Sigma_1, \Delta_1, \Delta_2)$, *unique*$(\Sigma_1, \Delta_1, \Delta_2)$ (5)

and $\llbracket t\rrbracket_{\Sigma_1, \Delta_1, \Delta_2} = (s, \Delta_3)$ (6)

By inversion of (1),

$\Sigma \vdash \Gamma'$ (7)

$\Sigma; \Gamma' \vdash t : \textbf{Kind}^{\textbf{P}}/\textbf{Kind}^{\textbf{T}}/\textbf{Prop}/\textbf{Type}$ (8)

By 3, (2), (3), (4), (5), (6), (8),

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1 \vdash^{CC} s : \Box/*$ (9)

and *unique*$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)$ (10)

By definition of *wf*, and (9),

wf$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1, x : s)$ (11)

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

case: $\mathcal{E}$ ends in T-PROP.

By assumption,

$$\mathcal{E} = \frac{\mathcal{E}' :: \Sigma \vdash \Gamma}{\Sigma; \Gamma \vdash \textbf{Prop} : \textbf{Kind}^{\textbf{P}}} \quad (1)$$

$[\![\Sigma]\!] = \Sigma_1$ (2)

$[\![\Gamma]\!]_{\Sigma_1} = (\Gamma_1, \Delta_1)$ (3)

$[\![\textbf{Prop}]\!]_{\Sigma_1, \Delta_1} = (*, \cdot)$ (4)

By *ax* rule,

$\cdot \vdash^{CC} * : \Box$ (5)

By 2, $\mathcal{E}'$, (2), (3),

wf$(\Sigma_1, \Delta_1, \Gamma_1)$ and *unique*(Σ_1, Δ_1) (6)

By Lemma weakening,

$\Sigma_1, \Delta_1, \Gamma_1 \vdash^{CC} * : \Box$ (7)

case: $\mathcal{E}$ ends in T-ARR.

By assumption,

$$\mathcal{E} = \frac{\mathcal{E}_1 :: \Sigma; \Gamma \vdash t_1 : (\textbf{Kind}^{\textbf{P}}, \textbf{Type}, \textbf{Prop}) \qquad \mathcal{E}_2 :: \Sigma; \Gamma, x : t_1 \vdash t_2 : k_2 \qquad k_2 \in \{\textbf{Kind}^{\textbf{P}}, \textbf{Prop}\}}{\Sigma; \Gamma \vdash (x{:}t_1) \rightarrow\ t_2 : k_2} \quad (1)$$

$[\![\Sigma]\!] = \Sigma_1$ (2)

$[\![\Gamma]\!]_{\Sigma_1} = (\Gamma_1, \Delta_1)$ (3)

wf$(\Sigma_1, \Delta_1, \Delta_2)$ and *unique*$(\Sigma_1, \Delta_1, \Delta_2)$ (4)

$[\![(x{:}t_1) \rightarrow\ t_2]\!]_{\Sigma_1, \Delta_1, \Delta_2} = ((x{:}s_1) \rightarrow\ s_2, (\Delta_3, \Delta_4))$ (5)

where $[\![t_1]\!]_{\Sigma_1, \Delta_1, \Delta_2} = (s_1, \Delta_3)$ (6)

and $[\![t_2]\!]_{\Sigma_1, \Delta_1, \Delta_2, \Delta_3} = (s_2, \Delta_4)$ (7)

By I.H. on $\mathcal{E}_1$,

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1 \vdash^{CC} s_1 : (*, \Box)$ (8)

and *unique*$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)$ (9)

By Definition of the translation of Γ, (3), (4),

$[\![\Gamma, x : t_1]\!]_{\Sigma_1} = ((\Gamma_1, x : s_1), (\Delta_1, \Delta_2, \Delta_3))$ (10)

By Lemma A.12, (8),

wf$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1)$ (11)

By I.H. on $\mathcal{E}_2$, (7), (10), (11), (9),

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Delta_4, \Gamma_1, x : s_1 \vdash^{CC} s_2 : (*/\Box)$ (12)

and *unique*$(\Sigma_1, \Delta_1, \Delta_3, \Delta_4)$ (13)

By Π, (8), (12),

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Delta_4, \Gamma_1 \vdash^{CC} (x{:}s_1) \rightarrow\ s_2 : (*/\Box)$ (14)

case: $\mathcal{E}$ ends in T-SIGN.

By assumption,

$$\mathcal{E} = \frac{\mathcal{E}_1 :: \Sigma \vdash \Gamma \qquad \Sigma; \cdot \vdash t_1 : \textbf{prin} \qquad \mathcal{E}_2 :: \Sigma; \cdot \vdash t_2 : \textbf{Prop}}{\Sigma; \Gamma \vdash \textbf{sign}(t_1, t_2) : t_1\ \textbf{says}\ t_2} \quad (1)$$

$[\![\Sigma]\!] = \Sigma_1$ (2)

$[\![\Gamma]\!]_{\Sigma_1} = (\Gamma_1, \Delta_1)$ (3)

wf$(\Sigma_1, \Delta_1, \Delta_2)$, *unique*$(\Sigma_1, \Delta_1, \Delta_2)$ (4)

$[\![\textbf{sign}(t_1, t_2)]\!]_{\Sigma_1, \Delta_1, \Delta_2} = (\mathit{fvar}, \Delta_3)$ (5)

where $[\![t_2]\!]_{\Sigma_1, \Delta_1, \Delta_2} = (s_2, \Delta_3)$ (6)

and $(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)(\mathit{fvar}) = s_2$ (7)

By I.H. on $\mathcal{E}_2$, (2), (4), (6),

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3 \vdash^{CC} s_2 : *$ (8)

and *unique*$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)$ (9)

By (7), (8), Lemma A.11 weakening,

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3 \vdash^{CC} \mathit{fvar} : s_2$ (10)

By 2 on $\mathcal{E}_1$, (2), (3),

wf$(\Sigma_1, \Delta_1, \Gamma_1)$ (11)

By Weakening and the domain of Γ_1 and Δ_2 Δ_3 are disjoint,

wf$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1)$ (12)

By Lemma A.11 weakening, (12), (10),

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1 \vdash^{CC} \mathit{fvar} : s_2$ (13)

case: $\mathcal{E}$ ends in T-SIGN.

By assumption,

$$\mathcal{E} = \frac{\mathcal{E}_1 :: \Sigma \vdash \Gamma \qquad \Sigma; \cdot \vdash t_1 : \textbf{prin} \qquad \mathcal{E}_2 :: \Sigma; \cdot \vdash t_2 : \textbf{Prop}}{\Sigma; \Gamma \vdash \textbf{sign}(t_1, t_2) : t_1 \ \textbf{says} \ t_2} \quad (1)$$

$\llbracket \Sigma \rrbracket = \Sigma_1$ (2)

$\llbracket \Gamma \rrbracket_{\Sigma_1} = (\Gamma_1, \Delta_1)$ (3)

wf$(\Sigma_1, \Delta_1, \Delta_2)$, *unique*$(\Sigma_1, \Delta_1, \Delta_2)$ (4)

$\llbracket \textbf{sign}(t_1, t_2) \rrbracket_{\Sigma_1, \Delta_1, \Delta_2} = (\mathit{fvar}_1, (\Delta_3, \mathit{fvar}_1 : s_2))$ (5)

where $\llbracket t_2 \rrbracket_{\Sigma_1, \Delta_1, \Delta_2} = (s_2, \Delta_3)$ (6)

and $\nexists \mathit{fvar}$ such that $(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)(\mathit{fvar}) = s_2$, and *fvar* is fresh (7)

By I.H. on $\mathcal{E}_2$, (2), (4), (6),

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3 \vdash^{CC} s_2 : *$ (8)

and *unique*$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)$ (9)

By *var* rule, (8),

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \mathit{fvar} : s_2 \vdash^{CC} \mathit{fvar} : s_2$ (10)

By (9), (7),

unique$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \mathit{fvar} : s_2)$ (11)

By 2 on $\mathcal{E}_1$, (2), (3),

wf$(\Sigma_1, \Delta_1, \Gamma_1)$ (12)

By Weakening and the domain of Γ_1 and Δ_2 $(\Delta_3, \mathit{fvar}_1 : s_2)$ are disjoint,

wf$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \mathit{fvar}_1 : s_2, \Gamma_1)$ (13)

By Lemma A.11 weakening, (13), (10),

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \mathit{fvar}_1 : s_2, \Gamma_1 \vdash^{CC} \mathit{fvar} : s_2$ (14)

case: $\mathcal{E}$ ends in T-APP RULE

By assumption,

$$\mathcal{E} = \frac{\mathcal{E}_1 :: \Sigma; \Gamma \vdash t_1 : (x{:}u_2) \rightarrow \ u \qquad \mathcal{E}_2 :: \Sigma; \Gamma \vdash t_2 : u_2}{\Sigma; \Gamma \vdash t_1 \ t_2 : \{t_2/x\}u} \quad (1)$$

$\llbracket \Sigma \rrbracket = \Sigma_1$ (2)

$\llbracket \Gamma \rrbracket_{\Sigma_1} = (\Gamma_1, \Delta_1)$ (3)

wf$(\Sigma_1, \Delta_1, \Delta_2)$, *unique*$(\Sigma_1, \Delta_1, \Delta_2)$ (4)

$\llbracket t_1 \ t_2 \rrbracket_{\Sigma_1, \Delta_1} = (s_1 \ s_2, (\Delta_2, \Delta_3, \Delta_4))$ (5)

and $\llbracket t_1 \rrbracket_{\Sigma_1, \Delta_1, \Delta_2} = (s_1, \Delta_3)$ (6)

and $\llbracket t_2 \rrbracket_{\Sigma_1, \Delta_1, \Delta_2, \Delta_3} = (s_2, \Delta_4)$ (7)

By I.H. on $\mathcal{E}_1$,

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1 \vdash^{CC} s_1 : k$ (8)

and $(k, \cdot) = \llbracket (x{:}u_2) \rightarrow \ u \rrbracket_{\Sigma_1, \Delta_1, \Delta_2, \Delta_3}$ (9)

and *unique*$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)$ (10)

By (8), and Lemma A.12,

wf$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1)$ (11)

By I.H. on $\mathcal{E}_2$, (10), (11), (7),

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Delta_4, \Gamma_1 \vdash^{CC} s_2 : k_2$ (12)

where $(k_2, \cdot) = [\![u_2]\!]_{\Sigma_1,\Delta_1,\Delta_2,\Delta_3,\Delta_4}$ (13)

and *unique*$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Delta_4$ (14)

By translation weakening Lemma A.10 and (9),

and $(k, \cdot) = [\![(x{:}u_2) \to \ u]\!]_{\Sigma_1,\Delta_1,\Delta_2,\Delta_3,\Delta_4}$ (15)

By definition of translation and (15), (13),

$k = (x{:}k_2) \to \ ku$ and $(ku, \cdot) = [\![u]\!]_{\Sigma_1,\Delta_1,\Delta_2,\Delta_3,\Delta_4}$ (16)

By *app* rule, (8), (12), (16),

$\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Delta_4, \Gamma_1 \vdash^{CC} s_1 \ s_2 : \{s_2/x\}ku$ (17)

By translation weakening Lemma A.10 and (7),

$[\![t_2]\!]_{\Sigma_1,\Delta_1,\Delta_2,\Delta_3,\Delta_4} = (s_2, \cdot)$ (18)

By Lemma A.14, (16), (18),

$[\![\{t_2/x\}u]\!]_{\Sigma_1,\Delta_1,\Delta_2,\Delta_3,\Delta_4} = (\{s_2/x\}ku, \cdot)$ (19)

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

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

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

Calculus of Construction extended with product dependent types is know to be strongly normalizing [22]. 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 strongly normalizing.

Lemma A.16 (Strong normalization of $\beta\beta'$-reduction in CC)**.** *For all term $t \in \mathbf{SN}(\beta)$, $t \in \mathbf{SN}(\beta\beta')$.*

Proof. We assign an ordering between terms as the dictionary order of a pair $(\beta(t), \delta(t))$, where $\beta(t)$ is the maximum *beta*-reduction steps of t, and $\delta(t)$ is defined as follows. $\delta(x) = 1$, $\delta(\lambda x{:}t.\, s) = \delta(s)$, $\delta(t_1\ t_2) = \delta(t_1) + 2\delta(t_2)$. We then prove that if $t \to_{\beta'} t'$ then $\beta(t') \le \beta(t)$, by examining all possible β-reductions of t', and showing that t has an corresponding reduction that takes at least the same number of β-reduction steps as t'. Now $\delta((\lambda y{:}s.\, ((\lambda x{:}t.\, t_1)t_2))u) = \delta(t_1) + 2\delta(t_2) + 2\delta(u)$, and $\delta(\lambda x{:}t.\, t_1)((\lambda y{:}s.\, t_2)u) = \delta(t_1) + 2\delta(t_2) + 4\delta(u)$. Therefore, when $t \to_{\beta'} t'$, $(\beta(t'), \delta(t')) < (\beta(t), \delta(t))$. Consequently, for all term $t \in \mathbf{SN}(\beta)$, $t \in \mathbf{SN}(\beta\beta')$. □

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

Lemma A.17 (Simulation)**.** *If $t \to t'$, and and $[\![t]\!]_\Delta = (s, \Delta)$, $[\![t']\!]_\Delta = (s', \Delta)$, then $s \to^{+}_{\beta,\beta'} s'$.*

Proof. By examining all the reduction rules. □

Lemma A.18 (Strong normalization)**.** AURA$_0$ *is strongly normalizing.*

Proof. By Lemma A.17, and Lemma A.16. 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.19. *If $s \to s'$, then $\{t/x\}s \to^* \{t/x\}s'$.*

Proof. By induction on the structure of s. □

Lemma A.20. *If $t \to t'$, then $\{t/x\}s \to^* \{t'/x\}s$.*

Proof. By induction on the structure of s. □

Lemma A.21 (Weak Confluence)**.** *If $t \to t_1$, $t \to t_2$, then exists t_3 such that $t_1 \to^* t_3$, and $t_2 \to^* t_3$.*

Proof. By induction on the structure of t. We invoke induction hypothesis directly in most of the cases. We show a few key cases below.

Case : $t = \lambda x{:}u.\, s$

By assumption,

$t_1 = \lambda x{:}u.\, s_1$ where $s \to s_1$ (1)

$t_2 = \lambda x{:}u.\, s_2$ where $s \to s_2$ (2)

By I.H. on s,

$\exists s_3$ such that $s_1 \to^* s_3$, and $s_2 \to^* s_3$ (3)

By reduction rules,

$t_3 = \lambda x{:}u.\, s_3$ such that $t_1 \to^* t_3$ and $t_2 \to^* t_3$ (4)

Case: $t = (\lambda x{:}u.\, s_1)s_2$, $t_1 = \{s_2/x\}s_1$, and
$t_2 = (\lambda x{:}u.\, s_1')s_2$ where $s_1 \to s_1'$.

By Lemma A.20,

$t_1 \to^* \{s_2/x\}s_1'$ (1)

By reduction rules,

$t_2 \to \{s_2/x\}s_1'$ (2)

Case: $t = \textbf{bind}\ x\ =\ s_1\ \textbf{in}\ s_2$
where $s_1 = \textbf{bind}\ y\ =\ \textbf{return}@[a]u_1\ \textbf{in}\ u_2$

By assumption,

$t_1 = \textbf{bind}\ x\ =\ \{u_1/x\}u_2\ \textbf{in}\ s_2$ (1)

$t_2 = \textbf{bind}\ y\ =\ \textbf{return}@[a]u_1\ \textbf{in}\ \textbf{bind}\ x\ =\ u_2\ \textbf{in}\ s_2$
and $y \notin \mathit{fv}(s_2)$ (2)

By reduction R-BINDT,

$t_2 \to \textbf{bind}\ x\ =\ \{u_1/y\}u_2\ \textbf{in}\ \{u_1/y\}s_2$ (3)

By (2), (3),

$t_2 \to \textbf{bind}\ x\ =\ \{u_1/y\}u_2\ \textbf{in}\ s_2$ (4)

Case: $t = \textbf{bind}\ x\ =\ s_1\ \textbf{in}\ s_2$
where $s_1 = \textbf{bind}\ y\ =\ u_1\ \textbf{in}\ u_2$
and $u_1 = \textbf{bind}\ z\ =\ w_1\ \textbf{in}\ w_2$

By assumption,

$t_1 = \textbf{bind}\ x\ =\ s_1'\ \textbf{in}\ s_2$
where $s_1' = \textbf{bind}\ z\ =\ w_1\ \textbf{in}\ \textbf{bind}\ y\ =\ w_2\ \textbf{in}\ u_2$ (1)
$t_2 = \textbf{bind}\ y\ =\ u_1\ \textbf{in}\ \textbf{bind}\ x\ =\ u_2\ \textbf{in}\ s_2$

By applying R-BINDC rule many times,

$t_1 \to^* t_3$ (2)

$t_2 \to^* t_3$ (3)

where $t_3 = \textbf{bind}\ z\ =\ w_1\ \textbf{in}\ \textbf{bind}\ y\ =\ w_2\ \textbf{in}\ \textbf{bind}\ x\ =\ u_2\ \textbf{in}\ s_2$

□