6-1-2009

Encoding Information Flow in AURA, Technical Appendix

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

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

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

Encoding Information Flow in AURA, Technical Appendix

Abstract

Two of the main ways to protect security-sensitive resources in computer systems are to enforce access-control policies and information-flow policies. In this paper, we show how to enforce information-flow policies in AURA, which is a programming language for access control. When augmented with this mechanism for enforcing information-flow polices, AURA can further improve the security of reference monitors that implement access control.

We show how to encode security types and lattices of security labels using AURA's existing constructs for authorization logic. We prove a noninterference theorem for this encoding. We also investigate how to use expressive access control policies specified in authorization logic as the policies for information declassification.

Comments

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

Encoding Information Flow in AURA, Technical Appendix

Department of Computer and Information Science
University of Pennsylvania
Technical Report Number MS-CIS-09-08
June, 2009

Limin Jia Steve Zdancewic
{liminjia, stevez}@seas.upenn.edu

Abstract

Two of the main ways to protect security-sensitive resources in computer systems are to enforce access-control policies and information-flow policies. In this paper, we show how to enforce information-flow policies in AURA, which is a programming language for access control. When augmented with this mechanism for enforcing information-flow polices, AURA can further improve the security of reference monitors that implement access control.

We show how to encode security types and lattices of security labels using AURA's existing constructs for authorization logic. We prove a noninterference theorem for this encoding. We also investigate how to use expressive access-control policies specified in authorization logic as the policies for information declassification.

1. Introduction

Almost all computer systems contain security-sensitive resources that need to be protected from untrusted applications. These include files, network connections, and private data such as a user's password or credit card number. Two of the main mechanisms for protecting these resources are access control and information-flow analysis. Access control aims to prevent unauthorized principals—human users or other computer systems—from gaining access to the resources. Enforcing information-flow policies focuses on protecting the confidentiality of private data and makes sure that attackers cannot guess secrets by observing the behavior of multiple runs of a program [22, 27]. In this paper, we investigate how to enforce information-flow policies in AURA [30, 17], a language for access control. When augmented with this mechanism for enforcing information-flow polices, AURA can further improve the security of reference monitors that implement access control.

We begin by a brief overview of information-flow analysis and AURA, a language for access control.

Enforcing information-flow polices To protect the confidentiality of information, researchers design advanced type systems to enforce that secret input data cannot be leaked by observation of a system's public output. The key idea in information-flow type systems (see the survey by Sabelfeld and Myers [27]) is that program data are given security types that indicate their security levels. For instance, if a secret integer is protected at security level H, we give this integer the type $\mathtt{int}_H$. The type system will guarantee that there is no information flow from high-security data to low-security data in well-typed programs. This property is referred to as noninterference.

However, information-flow policies that disallow any information flow from high security to low security are too draconian for real computer systems. Computer systems need to leak some amount of secret information to be useful. One classic example is the login program that compares user input with the password stored in the system, which is a secret. The boolean result of the comparison is not a secret because the login program has to either allow or deny access to the user. Therefore, attacker can always know if the password he typed in is the correct one or not. Another common example is that the average of all employees' salaries is released, but each individual salary has to be kept secret. Recently, there has been much work on controlled declassification of secret information [32, 23, 26, 18, 28, 10, 21, 6, 7, 8]. In the presence of declassification, the noninterference property does not hold.

AURA*, a language for access control* To ensure that only allowed principals can access protected resources, access-control requirements must be carefully defined and enforced. An *access-control policy* specifies whether a request by a principal to access a resource should be granted.

To clearly specify access-control policies and reason about them formally, researchers have developed authorization logics [5, 11, 14, 1, 2]. In logic-based access-control systems, logical proofs constructed using access-control policies serve as capabilities for accessing resources.

In these authorization logics, the formula A **says** P expresses principals' beliefs. A **says** P states that principal A believes that P is true. For instance, *Alice* **says** *SkyIsPurple* means that principal *Alice* believes that the sky is purple. *SkyIsPurple* is an assertion affirmed by a principal *Alice*. However, it is not necessarily the case that *SkyIsPurple* is true or that other principals believe it.

One desirable property of authorization logics is that principals should not interfere with each other's beliefs. Without explicit delegation, what a principal A believes should not be affected by other principals' beliefs. Such properties are also referred as noninterference properties [3, 15].

AURA is a language for implementing reference monitors for logic-based access control. AURA provides built-in support for specifying access-control policies. More specifically, the type system of AURA contains a constructive authorization logic based on DCC [2]. Programmers can manipulate authorization logic proofs as they do other language constructs. If implemented in AURA, a safe interface to access resources requires as an additional argument, a proof attesting that the access complies with the access-control policies. For example, a function *playFor*, which plays a song s on behalf of a principal p, might have the following type, which requires a proof that p is permitted to play s:
$(s:\mathit{Song}) \rightarrow (p:\textsf{prin}) \rightarrow \textsf{pf}\ (\textsf{self says}\ \mathit{MayPlay}\ p\ s) \rightarrow \mathit{Unit}.$

***Enforcing information-flow policies in* AURA** Our work is inspired by the work on building a library for light-weight information-flow security in Haskell [25]. In that work, information-flow types are encoded as a Haskell data type (Sec s t) where s is the security level. Sec is implemented as a monad and a module system guarantees that attackers cannot extract secrets hidden in the monad.

We use very similar high-level ideas to encode information-flow types in AURA. Our advantage over the Haskell approach is that we can use constructs for AURA's authorization logic for the encoding. The main idea of our encoding is that we use principals to represent security labels, and the type for a secret of type t protected at level H can be encoded as $(x : \textsf{pf}\ H\ \textsf{says}\ \mathit{Reveal}) \rightarrow t$. Intuitively, without H's private key, no one can create an assertion of the type H **says** *Reveal* and therefore secrets protected at level H can not flow to public channels.

The noninterference theorem of such encoding depends upon the noninterference properties of the authorization logic. Furthermore, expressive access-control policies specified in authorization logic can be used to specify the policies for declassification.

Contributions and roadmap This paper makes the following contributions.

- We show how to encode information-flow types using authorization logics based on prior work [30, 17].
- We prove the basic noninterference theorem of our encoding. The key components of the proof are mechanized in the proof assistant Coq [12].
- We investigate through examples how declassification can be governed by access-control policies.

The rest of the paper is organized as follows. In Section 2, we review AURA. In Section 3, we explain how to encode information-flow types using AURA's data types and the **says** monad. Next, in Section 4, we show how to prove the noninterference theorem for our encoding. Then, in Section 5, we extend our encoding and proof of noninterference to accommodate lattices of security labels. In Section 6, we investigate declassification. In the end, we discuss related work in Section 7.

2. AURA – A Language for Authorization and Audit

In this section, we give an overview of AURA to set up the background for the encoding of information-flow types in the next section. We will only discuss the high-level ideas. Technical details about the design of AURA can be found in our previous work [30, 17].

AURA is intended to be used to implement reference monitors for access control in security-sensitive settings. A reference monitor mediates access by allowing or denying requests to a resource (based, in this case, on policy specified in an authorization logic). For demonstrating key features of the language, we use an AURA implementation of a jukebox server as a running example.

2.1 Language Features

AURA is a call-by-value polymorphic lambda calculus. AURA consists of a "term-level" programming language for carrying out computation and a "proof-level" assertion language for writing proofs of access-control statements. AURA uses **Type** to classify the types of computations, and **Prop** to classify the types of proofs.

Authorization logic AURA allows programmers to define propositions like *MayPlay* using assertions. The following definition for *MayPlay* states that *MayPlay* takes a principal and a song as arguments and constructs a proposition.

$$\textsf{assert}\ \mathit{MayPlay} : \textsf{prin} \rightarrow \mathit{Song} \rightarrow \textsf{Prop}$$

While assertions are similar in flavor to datatypes with no constructors, there is a key difference: there is no pattern-matching statement associated with these assertions. Assertions such as *MayPlay* are only used as constants affirmed by principals to specify access-control polices.

In AURA, a **says** P is a proposition stating that principal a believes that proposition P is true. There are a few different ways to create a proof for a **says** P in AURA. We can construct a term of type a **says** P from a proof p of P using the operation **return** $a\ p$. We can also create the proof by chaining other proofs about a's beliefs using the bind operation written as (**bind** $x : Q = q$ **in** p). Here x stands in for the proof of Q encapsulated by q and p is a proof of a **says** P using x.

For example, consider the principals a and b, the song *freebird*, and the assertion *MayPlay* introduced earlier. The statements

$$\begin{array}{rcl} ok & : & a \ \textbf{says}\ (\mathit{MayPlay}\ a\ \mathit{freebird}) \\ \mathit{delegate} & : & b\ \textbf{says}\ ((p : \textbf{prin}) \rightarrow (s : \mathit{Song}) \rightarrow \\ & & \quad (a\ \textbf{says}\ (\mathit{MayPlay}\ p\ s)) \rightarrow \\ & & \quad (\mathit{MayPlay}\ p\ s)) \end{array}$$

assert that a gives herself permission to play *freebird* and b delegates to a the authority to allow other principals to play the song. These two terms may be used to create a proof of b **says** (*MayPlay a freebird*) as follows:

$$\begin{array}{l} \textbf{bind}\ d : ((p : \textbf{prin}) \rightarrow (s : \mathit{Song}) \rightarrow \\ \qquad (a\ \textbf{says}\ (\mathit{MayPlay}\ p\ s)) \rightarrow (\mathit{MayPlay}\ p\ s)) \\ \qquad = \mathit{delegate} \\ \textbf{in return}\ b\ (d\ a\ \mathit{freebird}\ ok). \end{array}$$

Such a proof could be passed to the *playFor* function if **self** is b, or it could be used to form a larger chain of reasoning.

In addition to uses of **return** and **bind**, AURA allows for the introduction of proofs of a **says** P without corresponding proofs of P by providing a pair of constructs, **say** and **sign**, that represent a principal's active affirmation of a proposition. The value **sign**(a, P) has type a **says** P; intuitively, we may think of it as a digital signature using a's private key on proposition P.

Only the principal a—or, equivalently, programs with access to a's private key—should be able to create a term of the form **sign**(a, P). We thus prohibit such terms from appearing in source programs and introduce the related term **say** P, which represents an effectful computation that uses the runtime's current authority—that is, its private key—to sign proposition P. When executed, **say** P generates a fresh value **sign**(**self**, P), where **self** is a built-in principal representing the current run-time authority.

It is worth noting that a principal can assert any proposition, even *False*. Because assertions are confined to the monad—thanks to the noninterference property of DCC—such an assertion can do little harm apart from making that particular principal's own assertions inconsistent.

Dependent types AURA incorporates dependent types: proofs in authorization logic can depend upon data, which allows for precise specification of access-control policies. For instance, the type of the proof that the *playFor* function requires is tied to the principal and the file arguments that *playFor* takes.

To simplify the meta-theory, AURA does not employ type-level reduction during type checking; and types only depend on values (i.e., well-formed normal forms). For instance, if S is a type constructor of the type $(x : Nat) \rightarrow$ **Type**, then $S\,(1+2)$ cannot be given a type in AURA because $1+2$ is not a value; but $S\,(1)$ has the type **Type**.

To make use of equalities obtained by run-time comparison of two values, AURA offers a type-refining equality test on *atomic* values—for instance, principals and booleans—as well as an explicit type cast between constructs of equivalent types. For example, when typechecking **if self** $=$ a **then** e_1 **else** e_2, the fact that **self** $= a$ is automatically made available while typechecking e_1 (due to the fact that **prin** is an atomic type). Therefore, in e_1 proofs of type **self says** P can be cast to type a **says** P and vice-versa.

The proof monad AURA uses the constant **pf** : **Prop** $\rightarrow$ **Type** to wrap access-control proofs as program values. Similar to the **says** monad, we can construct terms of the type **pf** P by using **return**$_\mathsf{p}$ p when p is a proof of P; or **bind**$_\mathsf{p}$ $x : t = q$ **in** p to chain proofs together [1].

Such a separation between proofs and computations is necessary to prevent effectful program expressions from appearing in a proof term. The type of **say** P is **pf** (**self says** P). If **say** P was given type **self says** P, it would be possible to create a bogus "proof" $\lambda x{:}\textbf{Prop}.\textbf{say}\ x$; the meaning of this "proof" would depend on the authority (**self**) of the program that applied the proof object.

Summary of syntax To simplify the presentation of AURA, it makes sense to unify as many of the constructs as possible. We thus adopt a lambda-cube style presentation [9] that uses the same syntactic constructs for terms, proofs, types, and propositions. A summary of AURA's core syntax is shown below.

$$\begin{array}{llcl} \text{Terms} & t & ::= & x \mid \mathit{ctr} \mid \ldots \\ & & \mid & \lambda x{:}t_1.t_2 \mid t_1\,t_2 \mid (x{:}t_1) \rightarrow t_2 \\ & & \mid & \textbf{match}\ t_1\ t_2\ \textbf{with}\ \{b\} \mid \langle t_1 : t_2 \rangle \\ \text{Branches} & b & ::= & \cdot \mid b \mid \mathit{ctr} \Rightarrow t \end{array}$$

In addition to the above common features (λ-abstraction, application, constructors, pattern matching, type cast, etc.), the AURA-specific syntax is shown below.

$$\begin{array}{lcl} t & ::= & \ldots \mid \textbf{Type} \mid \textbf{Prop} \mid \textbf{Kind} \mid \textbf{prin} \mid a\,\textbf{says}\,P \\ & \mid & \textbf{pf}\ P \mid \textbf{self} \mid \textbf{sign}(a, P) \mid \textbf{say}\ P \\ & \mid & \textbf{return}_\mathsf{s}\ a\ p \mid \textbf{bind}_\mathsf{s}\ x : t = e_1\ \textbf{in}\ e_2 \\ & \mid & \textbf{return}_\mathsf{p}\ p \mid \textbf{bind}_\mathsf{p}\ x : t = e_1\ \textbf{in}\ e_2 \\ & \mid & \textbf{if}\ v_1 = v_2\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \end{array}$$

AURA's value forms are as follows. We use metavariable v to denote values. We write $\mathit{val}(e)$ to mean that e is a value.

$$\begin{array}{lcl} v & ::= & x \mid \lambda x{:}t.e \mid \mathit{ctr}\,v_1 \cdots v_n \mid \textbf{self} \mid \textbf{sign}(v, p) \\ & \mid & \textbf{return}_\mathsf{s}\ v\ p \mid \textbf{bind}_\mathsf{s}\ x : t = p\ \textbf{in}\ q \mid \textbf{return}_\mathsf{p}\ v \end{array}$$

[1] In formal definitions, to distinguish the bind and return operation for **says** monad from those for **pf** monad, we annotate the bind and return with a subscript s for **says** monad and p for **pf** monad. However, the type checker can easily tell them apart; therefore, in AURA programs, bind and return are overloaded for both monads.

Signatures: data declarations and assertions Programmers can define bundles of mutually recursive datatypes and propositions in AURA just as they can in other programming languages. A signature *S* collects these data definitions and, as a consequence, a well-formed signature can be thought of as a map from constructor identifiers to their types.

For instance, we can define the boolean type as follows:

data *Bool* : **Type** {
| *tt* : *Bool*
| *ff* : *Bool*
}

Data definitions may be parametrized. For example, the familiar polymorphic list declaration is written as follows:

data *List* : **Type** $\to$ **Type** {
| *nil* : (t : **Type**) $\to$ *List t*
| *cons* : (t : **Type**) $\to t \to$ *List t* $\to$ *List t*
}

AURA's type system conservatively constrains **Prop** definitions to be inductive by disallowing negative occurrences of **Prop** constructors. Such a restriction is essential for consistency of the logic, since otherwise it would be possible to write loops that inhabit any proposition, including *False*.

2.2 Metatheory

The term typing judgment in AURA is written $S; E \vdash t : s$, where *S* is the signature containing definitions of data structures and assertions and *E* is the environment mapping variables to their types. We write $S \vdash \diamond$ to denote the well-formed judgments for signatures, and $S \vdash E$ to denote the well-formed judgments for environments. The small step operational semantics is denoted by $e \mapsto e'$.

We proved previously [17], the following properties of AURA. They will be useful in proving the noninterference properties of the information-flow type encoding in Section 4.

Theorem 1 (Preservation). *If* $S; \cdot \vdash e : t$ *and* $e \mapsto e'$*, then* $S; \cdot \vdash e' : t$.

Theorem 2 (Progress). *If* $S; \cdot \vdash e : t$ *then either* $val(e)$ *or exists* e' *such that* $e \mapsto e'$.

Theorem 3 (Typechecking is decidable).

- *If* $S \vdash \diamond$ *and* $S \vdash E$*, then* $\forall e, \forall t$*, it is decidable whether there exists a derivation such that* $S; E \vdash e : t$.
- *If* $S \vdash \diamond$ *then* $\forall E$ *it is decidable whether there exists a derivation such that* $S \vdash E$.
- *It is decidable whether there exists a derivation such that* $S \vdash \diamond$.

We also proved that the **Prop** fragment of AURA is strongly normalizing. This theorem will allow us to conclude that despite the intricate dependencies on data, the authorization logic fragment is still logically consistent.

In AURA's core language, the proofs are computation free, meaning that we do not have reduction rules on proofs. For instance, $\textbf{bind}_\textsf{s}\ x : t = p\ \textbf{in}\ q$ is a value. This is because the proofs are only meaningful as witnesses to access-control policies; and the reduction of proofs by reference monitors would not contribute significantly to the functionality of the system. We define proof reduction rules for the proofs in AURA, which will further reduce a "value" in the core language to a normal form according to the new reduction rules. We proved the following strong normalization theorem, details can be found in the appendix.

Theorem 4 (The proofs in AURA are strongly normalizing). *If* $S; \cdot \vdash e : P$*, and* $S; \cdot \vdash P :$ **Prop***, then e is strongly normalizing under the reduction rules for proofs.*

The noninterference proof also uses the following lemma stating that AURA's operational semantics is deterministic.

Lemma 5 (AURA's operational semantics is deterministic). *If* $e \mapsto^* v_1$ *and* $e \mapsto^* v_2$ *and* $val(v_1)$*,* $val(v_2)$ *then* $v_1 = v_2$.

3. Encoding Information Flow Types

In this section, we explain how to use AURA's authorization logic constructs to encode information-flow types. These types are indexed by the security level, at which data is protected. For lucid explanation of the main ideas, we assume there is only one security level *H* and all secrets are protected at level *H*. We will extend this encoding in Section 5 to accommodate standard lattices for security labels.

In our encoding, security labels as treated as principals. To support the definitions of security lattices (here the lattice only contains one security label), we extend AURA's signature to allow the definitions of constants of the type **prin** for declaring security labels. We can declare *H* as follows:

const *H* : **prin**

Next, we define the assertion *Reveal*.

assert *Reveal* : **Prop**

In this simple encoding, we use a value of the type **pf** (*H* **says** *Reveal*) as the capability to access secrets protected at level *H*. *Reveal* is the same kind of assertion as *MayPlay* shown in the previous section. In AURA, there is no term witnessing the proof of *Reveal*; therefore, a proof of *H* **says** *Reveal* can only be created by principal *H* actively affirming it by signing *Reveal* using its private key. Furthermore, we assume that *H* is not the run-time authority **self**, whose private key is the only private key that programmers have access to. With the above two conditions, we know that programmers cannot produce a term that is a proof of the proposition *H* **says** *Reveal*. We define a data type for secrets protected at level *H* below:

data *SecH* : **Type** $\to$ **Type** {
| *mkSec* : (t : **Type**)
$\to$ (**pf**(*H* **says** *Reveal*) $\to t$)
$\to$ *SecH t*
}

SecH is a polymorphic type constructor. For instance, *SecH Bool* is the type for boolean expressions protected at *H*. The data constructor *mkSec* takes two arguments. The first argument is a type *t*. The second argument is a function that when applied to a term of type **pf** (*H* **says** *Reveal*), yields the secret of type t. The secret data is in effect guarded by a capability of type **pf** (*H* **says** *Reveal*). For example, $s = \lambda x : \textsf{pf}\ (H\ \textsf{says}\ Reveal).3$ is a secret integer protected at level *H*. If there is a value v of the type **pf** $(H$ **says** $Reveal)$, evaluating $s\, v$ will reveal the secret 3.

A term e of the type **pf**(*H* **says** *Reveal*) belongs to the **Type** universe, meaning e is a computation. In AURA, programmers could write a non-termination computation Ω of the type **pf**(*H* **says** *Reveal*). However, this does not compromise our secret hidden in s, because AURA is call-by-value. Any attempt to execute $s\,\Omega$ and extract the term of type t from s will result in non-termination.

The only way to get hold of a value of type **pf**(*H* **says** *Reveal*) is when constructing another secret of type (*SecH s*) using *mkSec s* (λk : **pf** (*H* **says** *Reveal*).*e*). Here *k* is a capability for access secrets protected at *H*, and *k* is available in *e*. This means that terms of type *SecH t* operate like a monad; a computation that manipulates a secret has to have type *SecH t*. We can encode the standard return and bind operation for *SecH t* monad.

To create an expression of type *SecH t* from an expression of type *t*, we can use the following *Return* function.

```
Return : (t : Type) → (d : t) → (SecH t) =
  λt : Type. λd : t. mkSec t (λkey : pf(H says Reveal).d)
```

```
1  Bind : (t : Type) → (s : Type) → (d : SecH t)
2          → (f : t → SecH s) → SecH s
3   = λt : Type.λs : Type.λd : SecH t. λf : t → SecH s.
4  mkSec s
5       (λk : pf(H says Reveal).
6         (match d with {
7           | mkSec →
8             λdt : (pf(H says Reveal) → t).
9                match (f (dt k)) with{
10               | mkSec →
11                  λds : (pf(H says Reveal) → s).(ds k) }
12         }))
```

To operate on secrets, we can use the Bind function shown below. Given an expression *d* of type *SecH t*, and a function *f* that takes an expression of type *t* and produces an expression of type *SecH s*, *Bind* will apply *f* to the secrets in *d* and produce a term of type *SecH s*.

In the body of *Bind*, we need to apply function *f* (line 3) to the secret hidden in *d*. To extract the secret in *d*, we need a capability of type **pf** (*H* **says** *Reveal*). We can use such a capability *k* (line 5) in the body of the function we construct between line 5 and 12. We know *d* = *mkSec H dt* by pattern matching on *d* on line 6. The term *dt k* has type *t* because *dt* has type **pf** (*H* **says** *Reveal*) $\rightarrow$ *t* (line 8). *dt k* is the secret in *d*. The function application *f* (*dt k*) on line 9 has type *SecH s*. We need to construct a term of type *s*, because the function between line 5 and 12 has type **pf** (*H* **says** *Reveal*) $\rightarrow$ *s*. We pattern match on (*f* (*dt k*)) (line 9 – 11) and use *k* on line 11 to reveal the secret of type *s*.

Our encoding hides the expression that is a secret under a lambda abstraction, and because AURA does not evaluate under lambda abstractions, the computation in *SecH t* is lazy. A secret will not be evaluated until a capability for accessing the secret is provided.

4. Proof of Noninterference

To demonstrate that our encoding indeed protects secrets properly, we prove the noninterference theorem for our encoding. The main part of the proof is mechanized in Coq. The only paper proof is the proof of the noninterference property of AURA's authorization logic.

A noninterference proof is a proof of program equivalence. We want to prove that two programs containing different secrets should *behave the same* to the public observer. Here we use the termination-insensitive definition. We only enforce the equivalence between two programs when they both terminate. We use the squared semantics proof technique introduced by Pottier and Simonet [24]. The main idea of this approach is to define an extended language with a pair expression. The execution trace of a pair expression captures a pair of execution traces that could potentially contain different secrets. Proving the noninterference theorem is reduced to showing that two execution traces containing different secrets result in the same value in the extended language. Some of the challenges of using this techniques are 1) deciding where to introduce the pair expression so that the operational semantics can capture a pair of evaluation traces containing different secrets, and 2) introducing the pair expression in AURA in such a way that it works correctly with Aura's other language features such as dependent types.

The rest of this section is organized as follows. First, in Section 4.1, we introduce the design of AURA-PAIR, AURA extended with a pair construct. Next, in Section 4.2, we build connections between AURA and AURA-PAIR through a set of lemmas mapping the typing and evaluation relations between the two languages. Finally in Section 4.3, we discuss the proof of the noninterference theorem.

4.1 AURA-PAIR

We define AURA-PAIR by extending AURA with an expression denoting a pair of AURA expressions.

4.1.1 Syntax

We use meta-variables $\hat{t}$ and $\hat{e}$ to denote terms in AURA-PAIR, and use t and e to denote AURA terms. A summary of the syntax of AURA-PAIR is shown below. In addition to all the constructs in AURA, the definition of $\hat{t}$ includes a new construct $\langle t_1 \mid t_2 \rangle$. Since we syntactically require that t_1 and t_2 are terms from AURA, nested pair expressions are ruled out by this definition.

AURA-PAIR Terms

$$\begin{array}{lcl} \hat{t} & ::= & x \mid ctr \mid \lambda x{:}\hat{t}_1.\hat{t}_2 \mid \hat{t}_1\,\hat{t}_2 \mid (x{:}\hat{t}_1) \to \hat{t}_2 \\ & \mid & \cdots \mid \langle t_1 \mid t_2 \rangle \end{array}$$

AURA-PAIR Values

$$\hat{v} \;::=\; \lambda x{:}\hat{t}.\hat{v} \mid \cdots \mid \langle v_1 \mid v_2 \rangle$$

We also extend the values to include pair values, where each component in the pair is a value in AURA.

Before we define typing rules for AURA-PAIR, we introduce a few auxiliary definitions. First, we define a floor function that takes a term in AURA-PAIR and returns an AURA term that corresponds to either the left ($i = 1$) or the right ($i = 2$) part of the pair.

$$\boxed{\lfloor \hat{t} \rfloor_i = t} \quad \begin{array}{ll} \lfloor x \rfloor_i \;= x & \lfloor c \rfloor_i \;= c \\ \lfloor ctr \rfloor_i = ctr & \lfloor \hat{t}_1\,\hat{t}_2 \rfloor_i = \lfloor \hat{t}_1 \rfloor_i\,\lfloor \hat{t}_2 \rfloor_i \\ \cdots & \lfloor \langle t_1 \mid t_2 \rangle \rfloor_i = t_i \end{array}$$

For most constructs, the floor function is pushed into the sub-terms. For the pair expression, we return the subcomponents of the pair right away. For simplicity of presentation, we assume there is an implicit injection from an AURA term to an AURA-PAIR term. In our Coq proof, we defined such a function explicitly.

Since the pair expressions are not allowed to be nested in AURA-PAIR, we define a special capture-avoidance substitution for AURA-PAIR as follows.

$$\boxed{\hat{u}[\hat{t}/x] = \hat{u}'}$$

$$\begin{array}{l} (\hat{u}_1\,\hat{u}_2)[\hat{t}/x] \quad\;\; = (\hat{u}_1[\hat{t}/x])\,(\hat{u}_2[\hat{t}/x]) \\ \cdots \\ (\langle u_1 \mid u_2 \rangle)[\hat{t}/x] = \langle u_1\{\lfloor \hat{t} \rfloor_1/x\} \mid u_2\{\lfloor \hat{t} \rfloor_2/x\} \rangle \end{array}$$

For most cases, the substitution is standard. For the pair expression (last rule above), we use the term substitution in AURA, and substitute the floor of the term to be substituted ($\hat{t}$) for the variables in the sub-components of the pair (u_1 and u_2). Notice that $u_i\{\lfloor \hat{t} \rfloor_i/x\}$ is an AURA term. If we substitute an expression containing a pair into another pair expression, this substitution will make sure that the resulting expression does not contain nested pairs.

4.1.2 Operational Semantics

We use $\hat{e} \mapsto_p \hat{e}'$ to denote the small-step operational semantics of AURA-PAIR. Most evaluation rules are the same as the ones in AURA. The interesting reduction rules for AURA-PAIR are shown in Figure 1. For the APP rule, we use the special substitution defined above. Three additional rules are defined for evaluating the pair expression. The first two evaluate the terms inside a pair using the reduction rules in AURA. The last one lifts the pair when an application occurs. In Pottier and Simonet's original system, there is one lifting rule for each beta redex. We only have one such lifting rule for AURA-PAIR despite the fact that AURA has many beta redexes such as (**match** $(c\, v_1 \cdots v_n)$ t **with** $\{b\}$). The reason is that the typing judgments for AURA-PAIR restrict the appearance of the pair expression to function applications. This drastically simplifies the design of AURA-PAIR since we eliminated unnecessary lifting rules.

$$\frac{val(\hat{v})}{(\lambda x{:}\hat{t}.\hat{e})\,\hat{v} \mapsto_p \hat{e}[\hat{v}/x]}\;\textsc{App}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1 \mid e_2 \rangle \mapsto_p \langle e_1' \mid e_2 \rangle}\;\textsc{Pair-1} \qquad \frac{e_2 \mapsto e_2'}{\langle e_1 \mid e_2 \rangle \mapsto_p \langle e_1 \mid e_2' \rangle}\;\textsc{Pair-2}$$

$$\frac{val(v_1) \quad val(v_2) \quad val(\hat{v}_3)}{\langle v_1 \mid v_2 \rangle\,\hat{v}_3 \mapsto_p \langle v_1\,\lfloor \hat{v}_3 \rfloor_1 \mid v_2\,\lfloor \hat{v}_3 \rfloor_2 \rangle}\;\textsc{Lift}$$

Figure 1. Operational Semantics

4.1.3 Typing Rules

The typing judgment for AURA-PAIR is written $S; E \vdash^p \hat{e} : \hat{t}$. The only new typing rule is the rule for the pair expression, shown below. All other rules are the same as those in AURA.

$$\frac{\begin{array}{c} S \vdash^p E \quad S; \lfloor E \rfloor_i \vdash t_i : \lfloor (x{:}u_1) \to u_2 \rfloor_i \\ S; \cdot \vdash^p (x{:}u_1) \to u_2 : k \\ \nexists v \text{ such that } val(v) \text{ and } S; \cdot \vdash v : \lfloor u_1 \rfloor_i \end{array}}{S; E \vdash^p \langle t_1 \mid t_2 \rangle : (x{:}u_1) \to u_2}\;\textsc{Pair}$$

We assign an arrow type $(x : t_k) \to t$ to the pair expression, because the pair expression represents a pair of secrets, which have type (**pf** $(H$ **says** *Reveal*$) \to t$). The first argument of the arrow is the capability that cannot be forged. We enforce this by requiring that there is no value of such type under an empty context. Each sub-component of the pair is type checked under the floor of the result type. The floor operation is crucial for us because AURA is dependently typed, and the types may contain pair expressions as well.

It is strange to have a negation in the typing rules. The PAIR rule is still inductively defined because we are using the already-defined AURA's typing relation, and we have proven the decidability of the typing relation in AURA. Furthermore, this type system is never meant to be used to check programs. It is used to illustrate the noninterference properties of AURA. We do not have to consider the efficiency of using such a typing rule.

We proved progress and preservation theorems for AURA-PAIR. Since we already have Coq proofs for AURA, it was not too hard to change the proofs to prove the soundness of AURA-PAIR. In Pottier and Simonet's original paper, only preservation of the extended language is proven. The progress property simplifies the noninterference proof since we do not need to consider situations where AURA-PAIR might get stuck.

Theorem 6 (Preservation)**.** *If* $S; \cdot \vdash^p \hat{e} : \hat{t}$ *and* $\hat{e} \mapsto_p \hat{e}'$*, then* $S; \cdot \vdash^p \hat{e}' : \hat{t}$.

Theorem 7 (Progress)**.** *If* $S; \cdot \vdash^p \hat{e} : \hat{t}$ *then either* $val(\hat{e})$ *or exists* $\hat{e}'$ *such that* $\hat{e} \mapsto_p \hat{e}'$.

4.2 Connections Between AURA and AURA-PAIR

The point of defining AURA-PAIR is to compare two AURA programs. Here we establish the connection between programs in AURA and AURA-PAIR at both the typing and operational levels.

First, we establish the mapping between the special substitution in AURA-PAIR and the substitution in AURA.

Lemma 8 (Floor of Substitution)**.**
$\lfloor \hat{e}_2[\hat{e}_1/x] \rfloor_i = \lfloor \hat{e}_2 \rfloor_i \{\lfloor \hat{e}_1 \rfloor_i / x\}$

Lemmas 9 and 10 concern the mapping of typing relations between AURA and AURA-PAIR. Lemma 9 states that if an expression $\hat{e}$ is well-typed in AURA-PAIR, then both its left and right projection are well-typed in AURA. Lemma 10 states that a well-typed term in AURA is also well-typed in AURA-PAIR. We define $\lfloor E \rfloor_i$ to be the point-wise lifting of the floor function on the environment E.

Lemma 9 (Typing Soundness of AURA-PAIR)**.**
If $S; E \vdash^p \hat{e} : \hat{t}$ *then* $S; \lfloor E \rfloor_i \vdash \lfloor \hat{e} \rfloor_i : \lfloor \hat{t} \rfloor_i$.

Lemma 10 (Typing Completeness of AURA-PAIR)**.**
If $S; E \vdash e : t$ *then* $S; E \vdash^p e : t$.

The next two lemmas concern the evaluation behavior. The first lemma, Lemma 11, states that if a term $\hat{e}$ in AURA-PAIR evaluates to a value $\hat{v}$, then both the left and right projection in $\hat{e}$ should evaluate to values in AURA. This lemma tells us that AURA-PAIR adequately represents two traces of evaluation in AURA. The next lemma, Lemma 12, states that if both of the right and left projection of $\hat{e}$ evaluate to values in AURA, then $\hat{e}$ should evaluate to a value in AURA-PAIR. This lemma tells us that AURA-PAIR faithfully models the termination behavior of AURA.

Lemma 11 (Soundness of the Evaluation of AURA-PAIR)**.**
If $S; \cdot \vdash^p \hat{e} : \hat{t}$ *and* $\hat{e} \mapsto_p^* \hat{v}$ *then* $\lfloor \hat{e} \rfloor_i \mapsto^* \lfloor \hat{v} \rfloor_i$

Lemma 12 (Completeness of the Evaluation of AURA–PAIR)**.** *If* $S; \cdot \vdash^p \hat{e} : \hat{t}$ *and* $\lfloor \hat{e} \rfloor_i \mapsto^* v_i$ *where* v_i *is a value and* $i \in \{1, 2\}$ *then* $\exists \hat{u}$ *such that* $\hat{e} \mapsto_p^* \hat{u}$ *and* $\hat{u}$ *is a value.*

4.3 Noninterference

We use the following macros throughout this section.

$HKey = \textsf{pf}\ (H\ \textsf{says}\ Reveal) \quad SecHB = SecH\ Bool$

We define a function $CTROF(S, T)$ that takes a signature S and a type constructor T as arguments and returns the list of data constructors associated with T. For instance, $CTROF(S, Bool) = \{tt, ff\}$, if S contains the definition of *Bool*.

As we have mentioned in previous sections, the key idea of the encoding is to use *HKey* to guard secret data. We state this in the following lemma.

Lemma 13 (Secret)**.** $\nexists v, val(v)$ *and* $S; \cdot \vdash v : HKey$.

Proof. By contradiction. We use the strong normalization result of AURA, and the fact that programmers cannot generate the value $\textsf{sign}(H, Reveal)$.

Assume $S; \cdot \vdash v : HKey$
By Canonical Form, $HKey = \textsf{pf}\ (H\ \textsf{says}\ Reveal)$,
$\quad v = \textsf{return}_\textsf{p}\ q$ (1)
By Inversion of $S; \cdot \vdash v : HKey$,
$\quad S; \cdot \vdash q : H\ \textsf{says}\ Reveal$ (2)
By Strong Normalization results of AURA,
$\quad q \mapsto^* q'$ and q' is in normal form (3)
By Canonical Form, and $q' \neq \textsf{sign}(H, Reveal)$,
$\quad q' = \textsf{return}_\textsf{s}\ H\ c$ and $S; \cdot \vdash c : Reveal$ (4)
By Canonical Form,
$\quad c \in CTROF(S, Reveal) = \{\ \}$ (5)
Contradiction

□

The lemma assures us that no one can fabricate a value that has type *HKey*.

We prove the following noninterference theorem.

Theorem 14 (Noninterference)**.** *If* $S; x : SecHB \vdash e : Bool$ *and given any two values* v_1, v_2 *such that* $S; \vdash v_1 : SecHB$ $S; \vdash v_2 : SecHB$ *and* $e\{v_1/x\} \mapsto^* w_1$ *and* $e\{v_1/x\} \mapsto^* w_2$ *where* w_1, w_2 *are values, then* $w_1 = w_2$.

The proof is shown in Figure 2. To clearly present the structure of the proof, we write the proof in two columns. The left column contains statements in AURA and the right one contains statements in AURA-PAIR. The arrows between the two columns are labeled with lemmas from Section 4.2 that connect the properties of AURA and AURA-PAIR. The statements in gray boxes are assumptions of the noninterference theorem. The statement in the framed box is the conclusion.

The proof starts from the left column. First, we examine the values v_1 and v_2 and extract the sub-terms f_i, which contain secrets guarded by *HKey*. Next, using Lemma Secret (Lemma 13), we conclude that there is no value of type *HKey*, which allows us to go to the AURA-PAIR side and construct a value pair $\langle f_1 \mid f_2 \rangle$. Now the evaluation of expression $e[\hat{v}/x]$ captures the two evaluation traces containing different secrets. We stay on the AURA-PAIR side until we know that $e[\hat{v}/x]$ evaluates to a value $\hat{u}$ using the Evaluation Completeness Lemma (Lemma 12). Using the Evaluation Soundness Lemma, we go back to AURA and conclude that $e\{v_i/x\}$ evaluates to the floor of $\hat{u}$. Because AURA's reduction rules are deterministic (Lemma 5), we know that w_i is the same as the floor of $\hat{u}$. Now, we go to the AURA-PAIR side and gather more facts about $\hat{u}$. Because value $\hat{u}$ is of type *Bool*, we know that $\hat{u}$ has to be either the data constructor *tt* or *ff*. Because the floor of a constructor is itself, we know that both w_1 and w_2 have to be the same constructor.

5. Extension to Lattices

So far, we only considered single-level security where all secrets are protected at *H*. It is useful to have multi-level security where information is protected at several different security levels. For instance, a document could be classified

AURA		AURA-PAIR
(A1) $S; \cdot \vdash v_i : \mathit{SecHB}$ where $i \in \{1,2\}$		
By Canonical form lemma, A1	Typing rule PAIR $\rightarrow$	(B4) $S; \cdot \vdash^p \langle f_1 \mid f_2 \rangle : (x{:}\mathit{HKey}) \rightarrow \mathit{Bool}$
(A2) $\forall i \in \{1,2\}\ v_i = \mathit{mkSec\,Bool}\, f_i$		(B5) let $\hat{v} = \mathit{mkSec\,Bool}\, \langle f_1 \mid f_2 \rangle$
and $S; \cdot \vdash f_i : (x{:}\mathit{HKey}) \rightarrow \mathit{Bool}$		(B6) $S; \cdot \vdash^p \hat{v} : \mathit{SecHB}$
By Lemma 13(Secret)		(B8) $S; x : \mathit{SecHB} \vdash^p e : \mathit{Bool}$
(A3) $\nexists v, \mathit{val}(()v)$ and $S; \cdot \vdash v : \mathit{HKey}$		By Substitution Lemma, B5, B6, B8
		(B9) $S; \cdot \vdash^p e[\hat{v}/x] : \mathit{Bool}$
(A7) $S; x : \mathit{SecHB} \vdash e : \mathit{Bool}$	Lemma 10 (Typing Completeness) $\rightarrow$ (B8)	By Lemma 8 (Floor of Substitution) A2 B5
		(B10) $\lfloor e[\hat{v}/x] \rfloor_i = e\{v_i/x\}$
(A11) $e\{v_i/x\} \mapsto^* w_i$ where $i \in \{1,2\}$	Lemma 12 (Completeness) $\rightarrow$	(B12) $e[\hat{v}/x] \mapsto_p^* \hat{u}$ and $\hat{u}$ is a value
(A13) $e\{v_i/x\} \mapsto^* \lfloor \hat{u} \rfloor_i$ where $i \in \{1,2\}$	$\leftarrow$ Lemma 11 (Soundness)	By Preservation
By Lemma 5, A11, A13		(B15) $S; \cdot \vdash^p \hat{u} : \mathit{Bool}$
(A14) $\lfloor \hat{u} \rfloor_i = w_i$		By Canonical Form
(A17) $\boxed{w_1 = w_2 = \mathit{ctr}}$	$\leftarrow$ Use the definition of floor	(B16) $\hat{u} = \mathit{ctr}$, ($\mathit{ctr} = \mathit{tt}$ or ff)

Figure 2. Proof of non-interference theorem

as top secret, secret or public. We use a security lattice $\langle \mathcal{L}, \sqsubseteq \rangle$ to model multi-level security. $\mathcal{L}$ is a set of labels and $\sqsubseteq$ is a partial order on labels in $\mathcal{L}$. The information-flow policy captured by the security lattice is that if $\ell_1 \sqsubseteq \ell_2$, then information protected at ℓ_2 is more secret than information protected at ℓ_1, and information can only flow from ℓ_1 to ℓ_2.

We extend our encoding to enforce information-flow policies specified by security lattices. Throughout this section, we consider a two-point security lattice with labels: *H* and *L*, and the partial order between them: $L \sqsubseteq H$. The techniques for encoding the security lattice and proving noninterference can be carried over to handle more general security lattices.

5.1 Extended Encoding

Both H and L are constants of type **prin**. The partial order $L \sqsubseteq H$ is encoded using delegation in authorization logic as

L2H : *L* **says** (*H* **says** *Reveal* $\rightarrow$ *Reveal*)

In an implementation of the lattice in AURA, *L2H* can be the expression **sign**(L, H **says** *Reveal* $\rightarrow$ *Reveal*). It is an active affirmation by principal L by signing the proposition H **says** *Reveal* $\rightarrow$ *Reveal* using its private key.

Using *L2H* and hk : **pf** (H **says** *Reveal*), we can construct a term of the type **pf** (L **says** *Reveal*) as follows:

```
lk : pf (L says Reveal) =
    bind h : H says Reveal = hk in
    bind del : (H says Reveal → Reveal) = L2H in
    (return (return L (del h)))
```

Whenever we have a capability to reveal secrets protected at level *H*, we can obtain a capability to reveal secrets protected at level *L*.

We define the type constructor of security types below. The type constructor *Sec* takes as the first argument, the security level at which data is protected.

```
data Sec : prin → Type → Type {
 | mkSec : (l : prin) → (t : Type)
           → (pf (l says Reveal) → t)
           → Sec l t
}
```

Using the above definition, we can define the type for booleans protected at security level *L* as *Sec L Bool*, and the type for booleans protected at security level *H* as *Sec H Bool*.

The encodings of return and bind are similar to the ones in Section 3. The only difference is that we need to propagate the security label in the encoding.

```
Return : (l : prin) → (t : Type) → (d : t) → (Sec l t) =
 λl : prin.λt : Type. λd : t.
      mkSec t (λkey : pf(l says Reveal).d)

Bind : (l : prin) → (t : Type) → (s : Type) → (d : Sec l t)
         → (t → Sec l s) → Sec l s
 = λl : prin. λt : Type.λs : Type.λd : Sec l t.λf : t → Sec l s.
 match d with {
 | mkSec →
     λdt : (pf(l says Reveal) → t).
        mkSec l s (λkey : pf(l says Reveal).
               match (f (dt key)) with{
               | mkSec →
                 λds : (pf(l says Reveal) → s). ds key
               })
   }
```

We can treat secrets protected at level *L* as if they are protected at level *H*, since there are more information-flow restrictions on data protected at level *H* than at level *L*. We define a function *LtoH* that takes an expression of type *Sec L t* and return an expression of the type *Sec H t*.

```
LtoH : (t : Type) → Sec L t → Sec H t =
 λt : Type.λd : Sec L t.
  mkSec H t
    (λphk : pf(H says Reveal).
        match d with {
        | mkSec →
            λdl : (pf (L says Reveal) → t).
             dl (bind hk : (H says Reveal) = phk in
```

```
9          bind del : (H says Reveal → Reveal) = L2H
10         in return (return L (del hk)))
11       })
```

The body of *LtoH* changes the capability guarding the secret in d from **pf**(L **says** *Reveal*) to **pf**(H **says** *Reveal*). We start by using *mkSec* to construct a term protected at H. On line 4, the variable *phk* is the new capability associated with the secret data in d. The pattern-matching expression between line 5 and 11 constructs a term of type t by revealing the secret in d. To do so, we need a capability of the type **pf** (L **says** *Reveal*). We obtain such capability between line 8 and 10 by using *L2H*, which is the delegation from L to H and variable *phk*, which is the capability to access secrets protected at H (defined on line 4). The bind expression between line 8 and 10 is the same as *lk* we defined earlier in this section. The secret of type t hidden in d is revealed by applying *dl* (line 7) to the capability of type **pf** (L **says** *Reveal*), constructed at line 8– 10.

5.2 Noninterference

The encoding in Section 5.1 also has the noninterference property. Intuitively, by the noninterference properties of the authorization logic, we cannot prove H **says** *Reveal* from L **says** (H **says** *Reveal* → *Reveal*) and L **says** *Reveal*. Therefore, when constructing computations protected at the security level L, we cannot use any data protected at level H, which are guarded by a capability of type **pf**(H **says** *Reveal*).

Our proofs rely on the strong normalization results on AURA's authorization logic (Theorem 4). The idea is that any AURA term of type $H\,\textsf{says}\,\mathit{Reveal}$ can be normalized using proof reduction rules to a normal form, and we prove that no normal form has type $H\,\textsf{says}\,\mathit{Reveal}$.

We define the normal forms for proofs below.

Normal Forms

$$\begin{array}{rcl} \mathit{nf} & ::= & \lambda x\!:\!t_1.\mathit{nf} \mid c\,\mathit{nf}_1 \cdots \mathit{nf}_n \mid \textbf{return}_{\mathsf{p}}\,\mathit{nf} \\ & \mid & \textbf{return}_{\mathsf{s}}\,\mathit{nf}_1\,\mathit{nf}_2 \mid \textbf{sign}(\textbf{self}, \mathit{nf}) \\ & \mid & \textbf{bind}_{\mathsf{s}}\,x : t = \mathit{nf}_e\ \textbf{in}\ \mathit{nf} \mid \textbf{self} \\ & \mid & \textbf{bind}_{\mathsf{s}}\,x : t = \textbf{sign}(\textbf{self}, \mathit{nf})\ \textbf{in}\ \mathit{nf} \mid \mathit{nf}_e \\ & \mid & \textbf{Kind} \mid \textbf{Type} \mid \textbf{Prop} \mid \textbf{prin} \mid \textbf{pf}\,t \\ & \mid & (x\!:\!t_1) \rightarrow t_2 \end{array}$$

Elimination Normal Forms

$$\mathit{nf}_e \ ::= \ x \mid \mathit{nf}_e\,\mathit{nf} \mid \mathit{con}$$

The last two lines of the definition of *nf* are types. AURA has no reduction rules at the type level, so all the types are in normal form. The two $\textbf{bind}_{\mathsf{s}}$ expressions are stuck computations. Other stuck computations, such as $x\,y$, that are not $\textbf{bind}_{\mathsf{s}}$ expressions, are denoted by nf_e. We make a distinction between stuck computations that are $\textbf{bind}_{\mathsf{s}}$ expressions and those are not because we have a special commuting reduction rule on terms of the form $\textbf{bind}_{\mathsf{s}}\ x : u_1 = (\textbf{bind}_{\mathsf{s}}\ y : u_2 = t_1\ \textbf{in}\ t_2)\ \textbf{in}\ t_3$ see Figure 3 in the appendix for details .

The constants denoted by *con* include principals such as H and L defined for the lattice. We treat *L2H* that defines the partial order between L and H as a constant as well. This is because ordinary programmers cannot get hold of either H or L's private key, so the normal form of the programmer's code cannot include expressions of the form of $\textbf{sign}(H, P)$ or $\textbf{sign}(L, P)$. When treating *L2H* as an opaque constant, the definitions of *nf* and nf_e above generate the same normal form for programmers' code that makes use of *L2H* as a constant and that treats *L2H* as $\textbf{sign}(L, H\ \textsf{says}\,\mathit{Reveal} \rightarrow \mathit{Reveal})$.

We prove the following lemma, which is analogous to Lemma Secret (Lemma 13). This lemma assures us that we cannot construct a term witnessing H **says** *Reveal*, even if we assume L can make arbitrary assertions.

Lemma 15. *Secret H key*
if $\forall \mathit{con} \in \mathit{dom}(S)$, $S(x) = \textbf{prin}$, *or* $S(x) = L\,\textsf{says}\,t$,

- $\mathcal{E} :: S; \cdot \vdash \mathit{nf} : t$ *then* $t \neq H\,\textsf{says}\,\mathit{Reveal}$*, and* $t \neq \mathit{Reveal}$
- $\mathcal{E} :: S; \cdot \vdash \mathit{nf}_e : t$ *then* $t = L\,\textsf{says}\,P$ *or* $t = \textbf{prin}$

Proof (sketch): By mutual induction on derivation $\mathcal{E}$. □

The signature we care about is *SS*, which contains the definition of data type *Bool*, *Sec*, assertion *Reveal*, and constants representing the two-point security lattice.

$$\begin{array}{l} \mathit{SS} = \textbf{data}\ \mathit{Bool} : \textbf{Type}\ ..., \\ \quad \textbf{assert}\ \mathit{Reveal} : \textbf{Prop}, \\ \quad \textbf{data}\ \mathit{Sec} : \textbf{prin} \rightarrow \textbf{Type} \rightarrow \textbf{Type}\ ..., \\ \quad \textbf{const}\ H : \textbf{prin}, \textbf{const}\ L : \textbf{prin}, \\ \quad \textbf{const}\ \mathit{L2H} : L\ \textbf{says}\ (H\ \textbf{says}\ \mathit{Reveal} \rightarrow \mathit{Reveal}) \end{array}$$

As a corollary of Lemma 15, we can prove that we cannot construct a term of type **pf**(H **says** *Reveal*) from the lattice definitions and L **says** *Reveal*.

Lemma 16 (H Secret)**.**
$\nexists v, \mathit{val}(v)$ *and* $\mathit{SS}, \textbf{const}\ \mathit{LK} : L\,\textsf{says}\,\mathit{Reveal}; \cdot \vdash v : \mathit{HKey}$.

Proof (sketch): Using the strong normalization result and Lemma 15. □

We use the following macros for the rest of this section.

$$\begin{array}{rclcrcl} \mathit{LKey} & = & \textbf{pf}\ (L\,\textsf{says}\,\mathit{Reveal}) & \quad & \mathit{SecLB} & = & \mathit{Sec}\ L\ \mathit{Bool} \\ \mathit{HKey} & = & \textbf{pf}\ (H\,\textsf{says}\,\mathit{Reveal}) & \quad & \mathit{SecHB} & = & \mathit{Sec}\ H\ \mathit{Bool} \end{array}$$

This noninterference theorem below states that with two different secret inputs protected at security label H, the output values at level L are the same. The statement of the noninterference theorem with security lattices becomes more complicated because now we have to state that if two input values are the same for observers at security level L, then the output values of type *SecLB* are the same for observers at security level L. We indicate the presence of L observers by including a constant *LK* of the type (L **says** *Reveal*) in the signature for type checking the input values v_i. This is equivalent to saying that the observers at level L can see any secrets protect at level L, because $\textbf{return}_{\mathsf{p}}\,\mathit{LK}$ has type **pf** ($L\,\textsf{says}\,\mathit{Reveal}$). We cannot simply use the syntactic equality to state the equality of two values of type *SecLB*, because those values contain sub-terms that are functions. We need to specify that those functions evaluate to the same values when applied to the same arguments.

Theorem 17 (Noninterference)**.**
If $SS; x : SecHB \vdash e : SecLB$ *and given any two values* v_1, v_2 *such that* $SS, \textsf{const}\ LK : L\ \textsf{says}\ Reveal; \cdot \vdash v_i : SecHB$ *and* $e\{v_1/x\} \mapsto^* w_i$ *where* w_1, w_2 *are values, then* $w_i = mkSec\ L\ Bool\ f_i$ *and if* $(f_i\ (\textsf{return}_\textsf{p}\ LK)) \mapsto^* u_i$ *where* u_i *are values, then* $u_1 = u_2$.

The structure of the proof is very similar to the one shown in Figure 2. Due to space constraints, we omit the details. We explain two points in the proof: where Lemma Secret (Lemma 16) is used, and why the outputs are compared in the presence of *LK*.

In the proof, we know by the Canonical Forms Lemma that $v_i = mkSec\ H\ Bool\ g_i$. Lemma Secret (Lemma 16) allows us to construct a well-typed pair $\langle g_1 \mid g_2 \rangle$ in AURA-PAIR. This means that g_1 and g_2 are secrets given the current context and, therefore, could be put into a pair.

In the end, we know that $e[\hat{v}/x] \mapsto_p^* s$ and $w_i = \lfloor s \rfloor_i$. By canonical forms, we know that $s = mkSec\ L\ Bool\ q$, and $SS, \textsf{const}\ LK : L\ \textsf{says}\ Reveal; \cdot \vdash^p q : LKey \rightarrow Bool$. With LK in the signature, the canonical form will tell us that q has to be a lambda abstraction. Without LK, q itself could be a pair of functions. For observers at level *L*, q could not have been a pair because q does not contain information of higher secrecy than *L*.

6. Declassification

Information-flow polices that do not allow any information flow from high security to low security are typically too restrictive for practical use. To build useful systems, we often find it necessary to leak some amount of secret information. In this section, we explore through several examples the design space for using access-control policies to specify declassification policies in AURA.

6.1 Simple Declassification Policies

Escape hatches We can define a declassify operation similar to escape hatches [26]. The *declassify* function will reveal a secret protected at level *H*. If we assume that *declassify* is running under the authority *H*, the term **say** *Reveal* is a capability for revealing the secret, and we can implement *declassify* in AURA as follows.

$$
\begin{array}{l}
declassify : Sec\ H\ t \rightarrow Maybe\ t \\
= \lambda d : Sec\ H\ t. \\
\quad \textsf{match}\ d\ \textsf{with} \\
\quad \mid mkSec \rightarrow \\
\qquad \lambda dt : \textsf{pf}(H\ \textsf{says}\ Reveal) \rightarrow t. \\
\qquad \textsf{if}\ H = \textsf{self} \\
\qquad \textsf{then}\ Just\ (dt\ \langle(\textsf{say}\ Reveal) : \textsf{pf}\ (H\ \textsf{says}\ Reveal)\rangle) \\
\qquad \textsf{else}\ Nothing
\end{array}
$$

Since *H* is the same as **self**, in the true branch of the if expression we can use the explicit type cast to give the expression (**say** *Reveal*) the type **pf** (*H* **says** *Reveal*), which is used to reveal the secret hidden in *d*.

When More interestingly, we can use access-control policies to specify *when* information leaks are allowed. We can provide the following generic declassification interface:

$$declassify : Sec\ H\ t \rightarrow \textsf{pf}\ (H\ \textsf{says}\ Reveal) \rightarrow t$$

declassify takes two arguments: a secret protected at level *H* and the capability to reveal secrets protected at level *H*. *declassify* returns the secret hidden in the first argument.

We can define access-control policies that can be used to construct a proof of **pf** (*H* **says** *Reveal*). For instance, pol_1, below, specifies that if payment has been made, then the secret can be released. We use *Cashier* to represent the principal that controls the payment process. *Paid* is an assertion defined in the same way as *Reveal*.

$$pol_1 : H\ \textsf{says}\ (Cashier\ \textsf{says}\ Paid \rightarrow Reveal)$$

We can further define policies to specify when *Cashier* will affirm that payment has been made. For example, the following policy states that if *PNCBank* affirms that deposit has been made to account (Num), then *Cashier* will agree that payment has arrived.

$$
\begin{array}{l}
pol_c : Cashier\ \textsf{says}\ (PNCBank\ \textsf{says}\ (Deposited\ Num) \\
\qquad\qquad\qquad\qquad \rightarrow Paid)
\end{array}
$$

Alternatively, we could give the declassification interface the following more informative type.

$$declassify : Sec\ H\ t \rightarrow \textsf{pf}\ (Cashier\ \textsf{says}\ Paid) \rightarrow t$$

Who We can also specify to whom information is released. In the following example, we allow privileged users to access secret information. We use *Sys* to denote the principal System, who is in charge of deciding who are privileged principals. The predicate *Privileged p* means that principal *p* is a privileged principal and is defined below.

$$\textsf{assert}\ Privileged : \textsf{prin} \rightarrow \textsf{Prop}$$

Policy pol_2 allows the capability to access secrets protected at level *H* to be obtained by constructing a proof of *Sys* **says** (*Privileged p*).

$$pol_2 : H\ \textsf{says}\ (Sys\ \textsf{says}\ (Privileged\ p) \rightarrow Reveal)$$

The declassify interface that allows privileged principals to access secrets protected at *H* is shown below.

$$
\begin{array}{l}
declassify : (p : \textsf{prin}) \rightarrow Sec\ H\ t \\
\qquad\qquad \rightarrow Sys\ \textsf{says}\ (Privileged\ p) \rightarrow t
\end{array}
$$

The first argument of declassify is the principal to whom the information is released. The second argument is a secret protected at *H*. The third argument is a proof that *Sys* believes that *p* is a privileged user. The body of declassify uses pol_2 and returns the secret.

6.2 More Elaborate Policies

Refinement of secrets We can refine our encoding so that instead of using a single capability for all secrets, we can define different classes of secrets guarded by different capabilities. For example, the salaries of employees in the Engineer College are secrets. However, there are several different kinds of employees. We can use different capabilities to guard graduate students' salaries (**pf** (*H* **says** *GradSalary*)), postdocs' salaries (**pf** (*H* **says** *PostDocSalary*)) and professors' salaries (**pf** (*H* **says** *ProfSalary*)).

Here, *GradSalary*, *PostDocSalary* and *ProfSalary* are all assertions defined in the same way as *Reveal*. Now the security types need to indicate the class of secrets as well. For instance, instead of *Sec H t*, we use *Sec H Reveal t*.

We can write policies to declassify certain kinds of secrets. For example, the following statements declare that the Dean believes that postdocs and grad students are temporary employees. Predicate (*tmpE P*) means that *P* is a proposition used for guarding the salary info of temporary employees.

$$\begin{aligned} &s1 : \mathit{Dean}\ \textsf{says}\ (\mathit{tmpE}\ \mathit{PostDocSalary}) \\ &s2 : \mathit{Dean}\ \textsf{says}\ (\mathit{tmpE}\ \mathit{GradSalary}) \end{aligned}$$

The following declassification interface downgrades all information about temporary employees.

$$\begin{aligned} \mathit{declassify} : (R : \textsf{Prop})\ &\to \mathit{Sec}\ H\ R\ t \\ &\to \mathit{Dean}\ \textsf{says}\ (\mathit{tmpE}\ R)\ \to t \end{aligned}$$

Nonces One problem with the declassification interfaces shown so far is that a replay attack could cause unwanted information leaks. For instance, in the example where a proof of **pf** (*Cashier* **says** *Paid*) is required for the release of secrets, an attacker can use an old proof to learn all the secrets protected by **pf** (*H* **says** *Reveal*).

A standard way to prevent such replay attacks is to include a fresh nonce in the proofs, thereby preventing old proofs from being re-used. We can refine our encoding and include a nonce in the declassification interface.

$$\begin{aligned} &\textsf{data}\ \mathit{Nonce} : \textsf{Type}\{ \\ &\quad \mid n1 : \ \mathit{Nonce} \\ &\quad \ldots \\ &\quad \} \\ &\textsf{assert}\ \mathit{Paid} : \mathit{Nat}\ \to \textsf{Prop} \\ &\mathit{declassify} : (n : \mathit{Nonce})\ \to \mathit{Sec}\ H\ t \\ &\qquad\qquad \to \textsf{pf}\ (\mathit{Cashier}\ \textsf{says}\ (\mathit{Paid}\ n)) \\ &\qquad\qquad \to \mathit{Maybe}\ t \end{aligned}$$

A trusted nonce-generation function becomes part of the declassification interface. It produces a fresh nonce m for each declassification. In the body of the *declassify* function, the nonce a user passes in is checked against the stored current nonce for equality. Only when they are equal, the proof passed in by the user can be casted to a proof of **pf** (*Cashier* **says** (*Paid m*)), where m is the value of the stored current nonce. Therefore, an old proof with an expired nonce will not reveal the secret. We are in effect implementing a release-once policy.

However, the stored current nonce also becomes part of the trusted declassification interface. This means that implementing this version of the *declassify* function requires mutable state, which is not supported by AURA at this time.

6.3 Discussion

We have not studied the formal properties of these declassification policies. We suspect that the noninterference property of the authorization logic will allow us to express properties such as: "if a leak happens then certain principals must have made certain assertions". This kind of property would be useful for auditing purposes.

AURA does not support a module system or key management for run-time keys. Each process is associated with one run-time authority. This made it difficult to specify that *declassify* has to be run in trusted space and cannot be exploited by attackers. Furthermore, AURA's lack of support for mutable state prevents us from implementing declassification policies involving nonces, as shown in the example. If we were to add state to AURA, our encoding would need to be refined to consider possible information leaks from state changes. We speculate that techniques from prior work, such as those usef for building a library for light-weight information-flow security in Haskell [25] would apply. We leave these investigations for future work.

7. Related Work

Information flow type systems There has been much work on using language-based approaches to protect the confidentiality of information (Cf. [31, 16, 22, 33, 29]). Most of these works use security-label indexed types to indicate the security level of the data, and type systems enforce information-flow policies. Abadi *et al.* pointed out that information-flow analysis is a dependency analysis, and the noninterference property holds in the dependency core calculus (DCC) [4]. In DCC, security types are treated as security-label indexed monads. Abadi later showed that DCC can be used as a calculus for access control [2]. When DCC's monads are interpreted as principal-indexed types expressing principal's beliefs, DCC is isomorphic to an authorization logic. AURA contains a constructive authorization logic based on DCC in its type system [30, 17]. However, the principal-indexed monads in AURA cannot be used directly as security types. For example, we cannot use *H* **says** *int* as the type for an integer protected at level *H*. This is because AURA has two separate universes: one for logical proofs and propositions, which is pure; and the other for computations which may have effects such as non-termination. The separation is necessary for maintaining the soundness of AURA's authorization logic. The **says** monads are logical assertions, whereas the security types are the types for data and computations.

This paper provides an encoding of security types for data using the principal-indexed types.

Another approach to enforcing information-flow polices is to encode security types as libraries for existing functional languages, notably Haskell. Li *et al.* showed how to enforce information-flow policies in Haskell by encoding information-flow types using the arrow combinator [19]. A light-weight encoding of information-flow types using Haskell's type classes and abstract data types is later presented by Russo *et al.* [25]. Both of these encodings rely on Haskell's type classes and abstract data types to ensure that the information-flow policies are enforced. Our encoding relies on the noninterference properties of the authorization logic to enforce information-flow policies. One significant technical contribution of our work is that we proved a noninterference theorem for our encoding using the squared semantics approach [24], and that all aspects of our proofs related to the squared semantics are mechanized in Coq. To our knowledge, the noninterference proof in Russo's work [25] is done for an abstraction of the implementation. There is no formal proof about whether the abstraction faithfully models the implementation. Our proof of noninterference is done for the implementation itself, which had not been done. We acknowledge that Haskell is significantly more complicated than AURA, and our encoding does not consider side effects such as mutable references or IO, because AURA does not have these features. Another important contribution of our work is that we studied aspects of using access-control policies to declassify information.

Noninterference proofs of authorization logics The noninterference theorems of our encoding rely on the noninterference properties of AURA's authorization logic. We need to demonstrate that there is no value of the type $\textsf{pf}\ (H\ \textsf{says}\ \mathit{Reveal})$ (stated in Lemmas 13 and 16), which is a form of the noninterference property of the authorization logic in AURA. The first noninterference proof of a constructive authorization logic was done by Garg [15]. In Garg's proofs, the sub-formula properties of a cut-free sequent calculus are used to identify the assumptions that contribute to the conclusion. We could not use Garg's noninterference results directly because Garg's logic has different rules than ours, and it is first-order, while AURA's logic is second-order. In our proof, we use the strong-normalization results of the authorization logic in AURA. We examine all the possible normal forms of proofs that could be constructed using existing assumptions for encoding the lattice, and conclude that certain proofs are not possible. The statements of our noninterference theorem of the authorization logic (Lemmas 13 and 16) are not as general as Garg's. Encoding different lattices need different formulas, and we need to prove a lemma similar to Lemma 16 for each lattice. However, the techniques of our proofs are general enough for constructing proofs for other lattices. What's interesting in our work is that we demonstrate how to apply the noninterference property of an authorization logic to encoding information-flow types that have noninterference property.

Abadi also proved noninterference for CDD [3], a cut down version of DCC. However, in CDD, the lattice of principals does not correspond to delegation between them. We use explicit delegation between principals to encode lattices. Consequently, the noninterference proofs of CDD are not really applicable in our setting.

References

[1] M. Abadi. Logic in access control. In *Proceedings of the 18th Symposium on Logic in Computer Science (LICS)*, June 2003.

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

[3] M. Abadi. Access control in a core calculus of dependency. *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin ENTCS*, 172:5–31, April 2007.

[4] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, Jan. 1999.

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

[6] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

[7] A. Askarov and A. Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, 2007.

[8] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. *Security and Privacy, IEEE Symposium on*, 2008.

[9] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Clarendon Press, Oxford, 1992.

[10] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *In ESOP 2006: the 15th European Symposium on Programming*, 2006.

[11] J. Cederquist, R. Corin, M. Dekker, S. Etalle, and 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.

[12] The Coq Development Team. *The Coq Proof Assistant Reference Manual version 8.1*, 2007. Available from http://coq.inria.fr/.

[13] T. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris VII, 1985.

[14] H. DeYoung, D. Garg, and F. Pfenning. An authorization

logic with explicit time. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF-21)*, June 2008.

[15] D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, 2006.

[16] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, Jan. 1998.

[17] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit,. In *13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008.

[18] P. Li and S. Zdancewic. Downgrading Policies and Relaxed Noninterference. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 158 – 170, 2005.

[19] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, 2006.

[20] S. Lindley. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages*. PhD thesis, University of Edinburgh, College of Science and Engineering, School of Informatics, June 2005.

[21] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *In ESOP 2007: the 16th European Symposium on Programming*, 2007.

[22] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, Jan. 1999.

[23] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[24] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

[25] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, 2008.

[26] A. Sabelfeld and A. C. Myers. A Model for Delimited Release. In *International Symposium on Software Security*, 2003.

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

[28] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, 2005.

[29] V. Simonet. Flow Caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, Mar. 2003.

[30] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *Proc. of the IEEE Computer Security Foundations Symposium*, 2008.

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

[32] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Canada, June 2001.

[33] S. Zdancewic and A. C. Myers. Secure information flow and CPS. In *Proc. of the 10th European Symposium on Programming*, Apr. 2001.

Appendix

A. Summary of Typing Rules

AURA typing rules for standard functional language constructs.

$$\frac{S \vdash E}{S; E \vdash \textsf{Type} : \textsf{Kind}} \ \textsc{Wf-tm-type}$$

$$\frac{S \vdash E}{S; E \vdash \textsf{Prop} : \textsf{Kind}} \ \textsc{Wf-tm-prop}$$

$$\frac{S \vdash E \qquad S(ctr) = t}{S; E \vdash ctr : t} \ \textsc{Wf-tm-ctr}$$

$$\frac{S \vdash E \qquad E(x) = t}{S; E \vdash x : t} \ \textsc{Wf-tm-fv}$$

$$\frac{\begin{array}{c} S; E \vdash t_1 : k_1 \qquad S; E, x{:}t_1 \vdash t_2 : k_2 \\ k_2 \in \{\textsf{Type}, \textsf{Prop}, \textsf{Kind}\} \\ t_1 \in \{\textsf{Type}, \textsf{Prop}\} \text{ or } k_1 \in \{\textsf{Type}, \textsf{Prop}\} \end{array}}{S; E \vdash (x{:}t_1) \to t_2 : k_2} \ \textsc{Wf-tm-arr}$$

$$\frac{\begin{array}{c} S; E \vdash t : k \\ S; E, x{:}t \vdash u : k_1 \qquad S; E \vdash (x{:}u) \to k_1 : k_2 \\ t \in \{\textsf{Type}, \textsf{Prop}\} \text{ or } k \in \{\textsf{Type}, \textsf{Prop}\} \\ k_2 \in \{\textsf{Type}, \textsf{Prop}\} \end{array}}{S; E \vdash \lambda x{:}t.u : (x{:}t) \to k_1} \ \textsc{Wf-tm-abs}$$

$$\frac{\begin{array}{c} S; E \vdash t_1 : (x{:}u_2) \to u \qquad S; E \vdash t_2 : u_2 \\ S; E \vdash u_2 : k_2 \qquad S; E \vdash u\{x/t_2\} : ku \\ val(t_2) \\ \text{or} \\ (x \notin fv(u) \text{ and } (ku \in \{\textsf{Type}\} \text{ or } k_2 \in \{\textsf{Prop}, \textsf{Kind}\})) \end{array}}{S; E \vdash t_1\, t_2 : u\{x/t_2\}} \ \textsc{Wf-tm-app}$$

$$\frac{\begin{array}{c} S; E \vdash e : s \qquad s = ctr\ a_1\ a_2 \cdots a_n \\ S(ctr) = (x_1 : t_1) \to \cdots (x_n : t_n) \to u \\ branches\_cover\ S\ b\ ctr \\ S; E; s; (a_1, \cdots, a_n) \vdash b : t \\ S; E \vdash s : u \qquad S; E \vdash t : u \\ u \in \{\textsf{Type}, \textsf{Prop}\} \end{array}}{S; E \vdash \textsf{match}\ e\ t\ \textsf{with}\ \{b\} : t} \ \textsc{Wf-tm-matches}$$

AURA typing rules for access control constructs.

$$\frac{S \vdash E}{S; E \vdash \textsf{prin} : \textsf{Type}}\;\textsc{Wf-tm-prin}$$

$$\frac{S \vdash E}{S; E \vdash \textsf{self} : \textsf{prin}}\;\textsc{Wf-tm-self}$$

$$\frac{\begin{array}{c} S; E \vdash a : \textsf{prin} \\ S; E \vdash P : \textsf{Prop} \qquad \mathit{val}(a) \end{array}}{S; E \vdash a \,\textsf{says}\, P : \textsf{Prop}}\;\textsc{Wf-tm-says}$$

$$\frac{\begin{array}{c} S; E \vdash a : \textsf{prin} \qquad \mathit{val}(a) \\ S; E \vdash p : P \qquad S; E \vdash P : \textsf{Prop} \end{array}}{S; E \vdash \textsf{return}_{\textsf{s}}\ a\ p\ : a \,\textsf{says}\, P}\;\textsc{Wf-tm-says-ret}$$

$$\frac{\begin{array}{c} S; E \vdash e_1 : a \,\textsf{says}\, P \\ S; E, x : P \vdash e_2 : a \,\textsf{says}\, Q \\ x \notin \mathit{fv}(Q) \end{array}}{S; E \vdash \textsf{bind}_{\textsf{s}}\ x : P = e_1 \ \textsf{in}\ e_2 : a \,\textsf{says}\, Q}\;\textsc{Wf-tm-says-bind}$$

$$\frac{\begin{array}{c} S; \cdot \vdash a : \textsf{prin} \\ S; \cdot \vdash P : \textsf{Prop} \qquad \mathit{val}(a) \qquad S \vdash E \end{array}}{S; E \vdash \textsf{sign}(a, P) : a \,\textsf{says}\, P}\;\textsc{Wf-tm-sign}$$

$$\frac{S; E \vdash P : \textsf{Prop}}{S; E \vdash \textsf{say}\ P : \textsf{pf self says}\ P}\;\textsc{Wf-tm-say}$$

$$\frac{S; E \vdash P : \textsf{Prop}}{S; E \vdash \textsf{pf}\ P : \textsf{Type}}\;\textsc{Wf-tm-Pf}$$

$$\frac{S; E \vdash p : P \qquad S; E \vdash P : \textsf{Prop}}{S; E \vdash \textsf{return}_{\textsf{p}}\, p : \textsf{pf}\ P}\;\textsc{Wf-tm-pf-ret}$$

$$\frac{\begin{array}{c} S; E \vdash e_1 : \textsf{pf}\ P \\ S; E, x : P \vdash e_2 : \textsf{pf}\ Q \qquad x \notin \mathit{fv}(Q) \end{array}}{S; E \vdash \textsf{bind}_{\textsf{p}}\ x : P = e_1 \ \textsf{in}\ e_2 : \textsf{pf}\ Q}\;\textsc{Wf-tm-pf-bind}$$

$$\frac{\begin{array}{c} S; E \vdash v_1 : k \\ S; E \vdash v_2 : k \qquad \mathit{atomic}\ S\ k \qquad \mathit{val}(v_1) \\ \mathit{val}(v_2) \qquad S; E, x \sim (v_1 = v_2){:}k \vdash e_1 : t \\ S; E \vdash e_2 : t \qquad S; E \vdash k : \textsf{Type} \end{array}}{S; E \vdash \textsf{if}\ v_1 = v_2\ \textsf{then}\ e_1\ \textsf{else}\ e_2 : t}\;\textsc{Wf-tm-if}$$

$$\frac{\begin{array}{c} S; E \vdash e : s \\ \mathit{converts}\ E\ s\ t \qquad S; E \vdash t : \textsf{Type} \end{array}}{S; E \vdash \langle e : t \rangle : t}\;\textsc{Wf-tm-cast}$$

AURA typing rules for Branches

$\boxed{S; E; s; \mathit{args} \vdash \mathit{branches} : t}$

$$\frac{}{S; E; s; \mathit{args} \vdash \cdot : t}$$

$$\frac{\begin{array}{c} S; E; s; \mathit{args} \vdash b : t_r \quad S(c) = t_c \\ S; E \vdash \mathit{body} : t_b \qquad S; s; \mathit{args}; t_c; t_b; t_r \vdash \diamond \end{array}}{S; E; s; \mathit{args} \vdash b \mid c \ \Rightarrow \mathit{body} : t_r}$$

$\boxed{S; s; \mathit{args}; t_c; t_b; t_r \vdash \diamond}$

$$\frac{}{S; s; \cdot; s; t; t \vdash \diamond}$$

$$\frac{S; s; \cdot; t; u; k \vdash \diamond}{S; s; \cdot; (x : t_1) \to t; (x : t_1) \to u; k \vdash \diamond}$$

$$\frac{S; s; \mathit{args}; t\{a/x\}; u; k \vdash \diamond}{S; s; a, \mathit{args}; (x : t_1) \to t; u; k \vdash \diamond}$$

AURA environment typing rules

$\boxed{S \vdash E}$

$$\frac{}{S \vdash \cdot}\;\textsc{Wf-env-nil}$$

$$\frac{\begin{array}{c} S \vdash E \qquad S_1; E \vdash t : k \\ k \in \{\textsf{Type}, \textsf{Prop}\} \text{ or } t \in \{\textsf{Type}, \textsf{Prop}\} \\ x \text{ fresh} \end{array}}{S \vdash E, x : t}\;\textsc{Wf-env-cons-var}$$

$$\frac{\begin{array}{c} S \vdash E \qquad S; E \vdash t_1 : k \qquad S; E \vdash t_2 : k \\ \mathit{atomic}\ S\ k \qquad S; E \vdash k : \textsf{Type} \\ \mathit{val}(t_1) \qquad \mathit{val}(t_2) \qquad x \text{ fresh} \end{array}}{S \vdash E, x \sim (t_1 = t_2){:}k}\;\textsc{Wf-env-cons-eq}$$

AURA signature typing rules

$\boxed{S \vdash \mathsf{ok}}$

$$\frac{}{\cdot \vdash \mathsf{ok}}\;\textsc{Wf-sig-nil}$$

$$\frac{\begin{array}{l} S_1 = S\ \textsf{data}(T_1, i) : k_1\{un\} \\ \qquad \textsf{with} \cdots \\ \qquad \textsf{with data}(T_n, i) : k_m\{un\} \\ \Gamma_p = x_1 : s_1, \cdots x_p : s_p \\ \forall i \in [1, m], k_i = \forall \Gamma_p.\textsf{Type} \\ \forall j \in [1, in_i], S_1; \cdot \vdash t_{ij} : K \\ t_{ij} = \forall \Gamma_p . \forall \Gamma_a . (T_i\ x_1 \cdots x_p) \\ S_1 \vdash \diamond \\ T_i \notin \mathit{dom}(S) \quad c_{ij} \notin \mathit{dom}(S) \end{array}}{\begin{array}{l} S\ \textsf{data}(T_1, p) : k_1 \\ \qquad \{ \mid (c_{11} : t_{11}) \cdots \mid (c_{1n_1} : t_{1n_1}) \} \\ \ \textsf{with} \cdots \\ \ \textsf{with data}(T_m, p) : k_m \\ \qquad \{ \mid (c_{m1} : t_{m1}) \\ \qquad \cdots \\ \qquad \mid (c_{mn_m} : t_{mn_m}) \} \vdash \mathsf{ok} \end{array}}\;\textsc{Wf-bundle-Type}$$

$$\frac{\begin{array}{l} S_1 = S\ \textsf{data}(T_1, i) : k_1\{un\} \\ \quad \textsf{with} \cdots \\ \quad \textsf{with data}(T_n, i) : k_m\{un\} \\ \Gamma_p = x_1 : s_1, \cdots x_p : s_p \\ \forall i \in [1, m], k_i = \forall\Gamma_p.\textsf{Prop} \\ \forall j \in [1, in_i], S_1; \cdot \vdash t_{ij} : K \\ t_{ij} = \forall\Gamma_p.\forall\Gamma_a.(T_i\ x_1 \cdots x_p) \\ \textit{positive}\,\{T_1, \cdots, T_m\}\, t_{ij} \qquad S_1 \vdash \diamond \\ T_i \notin \textit{dom}(S) \quad c_{ij} \notin \textit{dom}(S) \end{array}}{\begin{array}{l} S\ \textsf{data}(T_1, p) : k_1 \\ \quad \{\mid(c_{11} : t_{11}) \cdots \mid(c_{1n_1} : t_{1n_1})\} \\ \textsf{with} \cdots \\ \textsf{with data}(T_m, p) : k_m \\ \quad \{\mid(c_{m1} : t_{m1}) \\ \quad \cdots \\ \quad \mid(c_{mn_m} : t_{mn_m})\} \vdash \textsf{ok} \end{array}} \text{ WF-BUNDLE-PROP}$$

$$\frac{\begin{array}{l} S \vdash \textsf{ok} \qquad S; \cdot \vdash t : \textsf{Kind} \\ t = (x_1 : t_1) \to \cdots (x_n : t_n) \to \textsf{Prop} \\ \textit{ctr} \notin \textit{dom}(S) \end{array}}{S\ \textsf{assert}\ \textit{ctr} : t \vdash \textsf{ok}} \text{ WF-ASSERT}$$

$\boxed{S \vdash \diamond}$

$$\frac{\begin{array}{l} S \vdash \textsf{ok} \\ \forall i \in (1, n), S; \cdot \vdash K_i : \textsf{Kind} \end{array}}{\begin{array}{l} S\ \textsf{data}(T_1, i) : K_1\{un\} \\ \textsf{with} \cdots \\ \textsf{with data}(T_n, i) : K_m\{un\} \vdash \diamond \end{array}} \text{ WF-TYPE-CONSTRUCTORS}$$

$$\frac{S \vdash \textsf{ok}}{S \vdash \diamond} \text{ WF-SIG-OK}$$

B. Strong Normalization

We prove the strong normalization result for AURA's authorization logic by translating proofs in AURA to CIC terms, which is believed to be strongly normalizing [13]. The typing rules for CIC terms that we refer to in the proofs can be found in [12].

B.1 Reduction Rules

We employ full-reduction on proofs. Because AURA is dependently typed, we need to make sure that proof reduction does not evaluate any expressions in **Type** universe which may diverge. Luckily, AURA's type system only allow values in **Type** to appear in proofs. However, reducing under lambda abstraction may lead to evaluation of expressions in **Type**. This is because $\lambda x{:}t.e$ is a value and it may appear in proofs. To avoid such situation, proof reduction rules need to distinguish between a lambda abstraction that is in **Type** from one in **Prop**. We do so by syntactically marking all the functions whose types are in **Type** before reducing *p*. During proof reduction, these marked terms do not reduce. This marking process is type-directed. We write $\lceil t\rceil_{S;E}$ to denote the resulting term of marking t under the context *S* and *E*. We use metavariable $\bar{t}$ to denote marked terms. We show some of the key marking rules below.

$\boxed{\lceil t\rceil_{S;E} = \bar{t}}$

$$\frac{}{\lceil x\rceil_{\text{-}} = x} \qquad \frac{}{\lceil \textsf{pf}\ P\rceil_{\text{-}} = \textsf{pf}\ \lceil P\rceil_{\text{-}}}$$

$$\frac{}{\lceil a\ \textsf{says}\ P\rceil_{S;E} = \lceil a\rceil_{S;E}\ \textsf{says}\ \lceil P\rceil_{S;E}}$$

$$\frac{}{\lceil \textsf{sign}(a, P)\rceil_{S;E} = \textsf{sign}(\lceil a\rceil_{S;E}, \lceil P\rceil_{S;E})}$$

$$\frac{}{\lceil \textsf{return}_\textsf{s}\ a\ p\ \rceil_{S;E} = \textsf{return}_\textsf{s}\ \lceil a\rceil_{S;E}\ \lceil p\rceil_{S;E}}$$

$$\frac{S;E \vdash \lambda x{:}t_1.t_2 : K \qquad S;E \vdash K : \textsf{Type}}{\lceil \lambda x{:}t_1.t_2\rceil_{S,E} = \boxed{\lambda x{:}t_1.t_2}}$$

$$\frac{S;E \vdash \lambda x{:}t_1.t_2 : K \qquad S;E \vdash K : \textsf{Prop}}{\lceil \lambda x{:}t_1.t_2\rceil_{S,E} = \lambda x{:}t_1.\lceil t_2\rceil_{S;E,x:t_1}}$$

For a lambda abstraction, if it has type **Type**, then we draw a box around it. Otherwise we mark the body of the function. Erasing all the boxes in a marked term converts it back to an unmarked term.

We define full reduction rules on proofs in Figure 3. Our proof reduction are defined over marked terms. There is no reduction rule for the boxed term.

For $\textsf{bind}_\textsf{s}$, in addition to the standard congruence, beta reduction, and commute rules as found in monadic languages, we also include a special beta reduction rule BINDS. The BINDS rule eliminates bound proofs that are never mentioned in the $\textsf{bind}_\textsf{s}$'s body. Rule BINDS permits simplification of terms like $\textsf{bind}_\textsf{s}\ x : u = \textsf{sign}(A, P)\ \textsf{in}\ t$, which are not subject to BINDBETA reductions. AURA disallows reduction under **sign**, as signatures are intended to represent fixed objects realized, for example, via cryptographic means.

We proved the following preservation theorem for the reduction rules.

Theorem 18 (Preservation)**.**
If $S;E \vdash p : P$ *and* $p \Longrightarrow p'$ *then* $S;E \vdash p' : P$.

Proof (sketch): By induction on the typing derivation $S;E \vdash p : P$. □

B.2 Translation

Term translation The translation of AURA terms is shown in Figure 4. The translation makes use of the following initial CIC signature Σ^0.

$$\begin{array}{rcl} \Sigma^0 & = & \textsf{TP} : \textsf{SET},\ \textsf{tm} : \textsf{TP}, \\ & & \textsf{PF\_RET} : \forall P{:}\textsf{PROP}.\forall x{:}P.\textsf{TP}, \\ & & \textsf{SIGN} : \forall P{:}\textsf{PROP}.P \\ & & \textsf{Ind}(\cdot)\textsf{FF} : \textsf{Set} := \{\},\ \textsf{ff} : \textsf{FF} \end{array}$$

$$\boxed{\bar{t} \Longrightarrow \bar{t}'}$$

$$\frac{}{(\lambda x\!:\!t_1.\bar{t}_2)\,\bar{t}_3 \Longrightarrow \bar{t}_2\{\bar{t}_3/x\}}\ \text{APPBETA}$$

$$\frac{(C_i\,\bar{t}_1\,\cdots\bar{t}_n, C_i, body) \Longrightarrow_c (\bar{t}', 0)}{\textsf{match}\ (C_i\,\bar{t}_1\,\cdots\bar{t}_n)\ k\ \textsf{with}\ \{\mid\cdots\mid C_i\ \Rightarrow body\cdots\} \Longrightarrow \bar{t}'}\ \text{MATCHBETA}$$

$$\frac{}{\textsf{bind}_\textsf{s}\ x : u = (\textsf{return}_\textsf{s}\ \bar{t}\ \bar{t}_1\ )\ \textsf{in}\ \bar{t}_2 \Longrightarrow \bar{t}_2\{\bar{t}_1/x\}}\ \text{BINDBETA}$$

$$\frac{x \notin fv(\bar{t}_2)}{\textsf{bind}_\textsf{s}\ x : u = \bar{t}_1\ \textsf{in}\ \bar{t}_2 \Longrightarrow \bar{t}_2}\ \text{BINDS}$$

$$\frac{y \notin fv(\bar{t}_3)}{\textsf{bind}_\textsf{s}\ x : u_1 = (\textsf{bind}_\textsf{s}\ y : u_2 = \bar{t}_1\ \textsf{in}\ \bar{t}_2)\ \textsf{in}\ \bar{t}_3 \Longrightarrow \textsf{bind}_\textsf{s}\ y : u_2 = \bar{t}_1\ \textsf{in}\ (\textsf{bind}_\textsf{s}\ x : u_1 = \bar{t}_2\ \textsf{in}\ \bar{t}_3)}\ \text{BINDC}$$

$$\frac{\bar{t}_1 \Longrightarrow \bar{t}_1'}{\textsf{bind}_\textsf{s}\ x : u = \bar{t}_1\ \textsf{in}\ \bar{t}_2 \Longrightarrow \textsf{bind}_\textsf{s}\ x : u = \bar{t}_1'\ \textsf{in}\ \bar{t}_2}\ \text{BIND1}$$

$$\frac{\bar{t}_2 \Longrightarrow \bar{t}_2'}{\textsf{bind}_\textsf{s}\ x : u = \bar{t}_1\ \textsf{in}\ \bar{t}_2 \Longrightarrow \textsf{bind}_\textsf{s}\ x : u = \bar{t}_1\ \textsf{in}\ \bar{t}_2'}\ \text{BIND2}$$

$$\frac{\bar{t}_2 \Longrightarrow \bar{t}_2'}{\lambda x\!:\!t_1.\bar{t}_2 \Longrightarrow \lambda x\!:\!t_1.\bar{t}_2'}\ \text{LAM}$$

$$\frac{\bar{t}_2 \Longrightarrow \bar{t}_2'}{\textsf{return}_\textsf{s}\ \bar{t}_1\ \bar{t}_2 \Longrightarrow \textsf{return}_\textsf{s}\ \bar{t}_1\ \bar{t}_2'}\ \text{SAYS}$$

$$\frac{\bar{t}_1 \Longrightarrow \bar{t}_1'}{\bar{t}_1\,\bar{t}_2 \Longrightarrow \bar{t}_1'\,\bar{t}_2}\ \text{APP1} \qquad \frac{\bar{t}_2 \Longrightarrow \bar{t}_2'}{\bar{t}_1\,\bar{t}_2 \Longrightarrow \bar{t}_1\,\bar{t}_2'}\ \text{APP2}$$

$$\frac{\bar{t} \Longrightarrow \bar{t}'}{\textsf{match}\ \bar{t}\ k\ \textsf{with}\ \{b\} \Longrightarrow \textsf{match}\ \bar{t}'\ k\ \textsf{with}\ \{b\}}\ \text{MATCH}$$

$$\boxed{(\bar{t}, c, body) \Longrightarrow_c (\bar{t}, n)}$$

$$\frac{}{((c,n),(c,n),body) \Longrightarrow_c (body, n)}\ \text{CTR-BASE}$$

$$\frac{m > 0 \qquad (\bar{t}_1,(c,n),body) \Longrightarrow_c (body, m)}{(\bar{t}_1\,\bar{t}_2,(c,n),body) \Longrightarrow_c (body, m-1)}\ \text{CTR-PARAM}$$

$$\frac{(\bar{t}_1,(c,n),body) \Longrightarrow_c (\bar{t}, 0)}{(\bar{t}_1\,\bar{t}_2,(c,n),body) \Longrightarrow_c (\bar{t}\,\bar{t}_2, 0)}\ \text{CTR-ARG}$$

Figure 3. Proof Reduction Rules

Since we need to distinguish terms in **Type** from terms in **Prop**, the translation is directed by the typing judgment. We translate **Type** to SET, **Prop** to PROP, and **Kind** to TYPE. Because the translation only need to preserve well-typedness and evaluation behavior, we can safely translate the type of **pf** *P* to TP. We erase monads and translate the bind expression to function application. We translate $\textsf{sign}(a, P)$ using the constant SIGN defined in Σ^0. Similarly for $\textsf{return}_\textsf{p}\, p$, we translate it using PF_RET in Σ^0. The lambda abstraction $\lambda x\!:\!t.e$ in the **Type** universe is translated to a pattern-matching term on ff, which is a constant of type FALSE. As a result, this pattern match can be given any type. The translation of application $t_1\,t_2$ does not consider the case where $t_1\,t_2$ has type in **Type** and is not a value. For instance $(\lambda x\!:\!t_1.t_2)\,t_3$ of type **bool** does not have a translation, because it could be a diverging computation.

Environment translation The translation of the typing environment *E* is shown below. Notice that we only translate environments with variable type bindings and ignore equality bindings. This is sound because the if statement is not a value, and we do not need to translate it. This means that the environment *E* during translation does not contain equality bindings.

$$\boxed{[\![E]\!]_S = \Gamma}$$

$$\frac{}{[\![\cdot]\!]_S = \cdot} \qquad \frac{}{[\![E, x : t]\!]_S = [\![E]\!]_S, x : [\![t]\!]_{S,E}}$$

Signature translation The translation for signatures is show in Figure 5. We translate AURA's datatype definitions in the **Prop** universe to CIC's inductive data types. AURA's datatype definitions that are in the **Type** universe allow recursive datatypes that are not allowed in CIC. Therefore, we translate those definitions into CIC constants. This is sound because we do not need to translate any pattern-matching statement that examines an expression whose type is in the **Type** universe.

B.3 Strong Normalization Proofs

Throughout this section, we write *constructs* $t\ K$ to denote $t = (x_1\!:\!t_1) \to \cdots (x_n\!:\!t_n) \to K$.

Lemma 19 (Translation Weakening Context)**.**
If $[\![e]\!]_{S;E_1,E_3} = t$*, and* $S \vdash E_1, E_2, E_3$*,*
then $[\![e]\!]_{S;E_1,E_2,E_3} = t$*.*

Proof (sketch): By induction on $[\![e]\!]_{S;E_1,E_3} = t$. □

Lemma 20 (Translation Weakening Signature (Undefined))**.**

If $\mathcal{E} :: [\![e]\!]_{S,\overline{(T_i,k_i,un)};E} = t$*, and* $S, \overline{(T_i, k_i, cdel_i)} \vdash \diamond$*, then* $[\![e]\!]_{S,\overline{(T_i,k_i,cdel_i)};E} = t$*.*

Proof (sketch): By induction on the derivation $\mathcal{E}$, and use weakening properties of AURA directly. □

Lemma 21 (Translation Weakening Signature)**.**
If $\mathcal{E} :: [\![e]\!]_{S_1;E} = t$*, and* $S_1, S_2 \vdash \diamond$*, then* $[\![e]\!]_{S_1,S_2;E} = t$*.*

$\boxed{[\![t]\!]_{S,E} = s}$

$$\frac{}{[\![\textsf{Kind}]\!]_{\_} = \mathsf{TYPE}} \qquad \frac{}{[\![\textsf{Type}]\!]_{\_} = \mathsf{SET}}$$

$$\frac{}{[\![\textsf{Prop}]\!]_{\_} = \mathsf{PROP}} \qquad \frac{}{[\![x]\!]_{\_} = x} \qquad \frac{}{[\![c]\!]_{S,\_} = c}$$

$$\frac{}{[\![\textsf{self}]\!]_{\_} = \mathsf{tm}} \qquad \frac{}{[\![\textsf{prin}]\!]_{\_} = \mathsf{TP}} \qquad \frac{}{[\![\textsf{pf}\ P]\!]_{\_} = \mathsf{TP}}$$

$$\frac{}{[\![a\ \textsf{says}\ P]\!]_{S;E} = [\![P]\!]_{S;E}}$$

$$\frac{}{[\![\textsf{sign}(a,P)]\!]_{S;E} = \mathsf{SIGN}\ [\![P]\!]_{S;E}}$$

$$\frac{}{[\![\textsf{return}_{\mathsf{s}}\ a\ p\ ]\!]_{S;E} = [\![p]\!]_{S;E}}$$

$$\frac{}{[\![\textsf{bind}_{\mathsf{s}}\ x : u = t_1\ \textsf{in}\ t_2]\!]_{S;E} = (\lambda x\!:\![\![u]\!]_{S;E}.[\![t_2]\!]_{S;E})\ [\![t_1]\!]_{S;E}}$$

$$\frac{S;E \vdash p : P \qquad [\![p]\!]_{S;E} = q \qquad Q = [\![P]\!]_{S;E}}{[\![\textsf{return}_{\mathsf{p}}\ p]\!]_{S;E} = \mathsf{PF\_RET}\ Q\ q}$$

$$\frac{s_1 = [\![t_1]\!]_{S;E} \qquad s_2 = [\![t_2]\!]_{S;E,x:t_1}}{[\![(x\!:\!t_1) \to t_2]\!]_{S;E} = (x\!:\!s_1) \to s_2}$$

$$\frac{S;E \vdash \lambda x\!:\!t_1.t_2 : K \qquad S;E \vdash K : \textsf{Type}}{[\![\lambda x\!:\!t_1.t_2]\!]_{S,E} = \textsf{match}\ \mathsf{ff}\ \textsf{return}\ [\![K]\!]_{S,E}\ \textsf{with}\ \{\}}$$

$$\frac{\begin{array}{c} S;E \vdash \lambda x\!:\!t_1.t_2 : K \\ S;E \vdash K : \textsf{Prop} \qquad s_1 = [\![t_1]\!]_{S;E} \qquad s_2 = [\![t_2]\!]_{S;E,x:t_1}\end{array}}{[\![\lambda x\!:\!t_1.t_2]\!]_{S,E} = \lambda x\!:\!s_1.s_2}$$

$$\frac{\begin{array}{c} \mathit{val}(t_1\,t_2) \\ \text{or } S;E \vdash t_1\,t_2 : K \qquad S;E \vdash K : S \text{ and } S \neq \textsf{Type}\end{array}}{[\![t_1\,t_2]\!]_{S;E} = [\![t_1]\!]_{S;E}\,[\![t_2]\!]_{S;E}}$$

$$\frac{S;E \vdash e : s,\ S;E \vdash s : \textsf{Prop}}{\begin{array}{c}[\![\textsf{match}\ e\ t\ \textsf{with}\ \{\,|\,b_1 \cdots |\,b_n\}]\!]_{S;E} = \\ \textsf{match}\ [\![e]\!]_{S;E}\ \mathsf{return}\ [\![t]\!]_{S;E}\ \textsf{with}\ \{\,|\,[\![b_1]\!]_{S;E} \cdots |\,[\![b_n]\!]_{S;E}\}\end{array}}$$

Figure 4. Translation of AURA terms

Proof (sketch): By induction on the derivation $\mathcal{E}$, and use weakening properties of AURA directly. □

Lemma 22 (Typing Inversion)**.** *If* $S;E \vdash e : t$ *and* $S;E \vdash t : k$*, then* $k = \textsf{Type}, \textsf{Prop}, \textsf{Kind}$.

Proof (sketch): By induction on the typing derivation $S;E \vdash e : t$. □

Lemma 23 (Constructs Substitution)**.**

$\boxed{[\![S]\!] = \Sigma(\Gamma)}$

$$\frac{}{[\![\cdot]\!] = \cdot}$$

$$\frac{k_j = (x_1\!:\!t_1) \to \cdots (x_n\!:\!t_n) \to \textsf{Prop}}{\begin{array}{l}[\![S \\ \ \textsf{data}(T_1,p) : k_1\{\,|\,(c_{11} : t_{11}) \cdots |\,(c_{1n_1} : t_{1n_1})\} \\ \ \textsf{with} \cdots \\ \ \textsf{with data}(T_m,p) : k_m\{\,|\,(c_{m1} : t_{m1}) \cdots |\,(c_{mn_m} : t_{mn_m})\}]\!] \\ = [\![S]\!], \mathsf{Ind}(\cdot)[\mathsf{p}](\mathsf{T_1} : [\![\mathsf{k_1}]\!]_S, \cdots, \mathsf{T_m} : [\![\mathsf{k_m}]\!]_S) := \\ \quad (c_{11} : [\![t_{11}]\!]_S), \cdots (c_{1n_1} : [\![t_{1n_1}]\!]_S), \cdots, \\ \quad (c_{m1} : [\![t_{m1}]\!]_S), \cdots (c_{mn_m} : [\![t_{mn_m}]\!]_S)\end{array}}$$

$$\frac{K_j = (x_1\!:\!t_1) \to \cdots (x_n\!:\!t_n) \to \textsf{Prop}}{\begin{array}{l}[\![S\ \textsf{data}(T_1,i) : K_1\{un\}\ \textsf{with} \cdots\ \textsf{with data}(T_n,i) : K_m\{un\}]\!] \\ = [\![S]\!], (T_1 : [\![k_1]\!]_S, \cdots, T_m : [\![k_m]\!]_S)\end{array}}$$

$$\frac{}{[\![S, \textsf{assert}\ c : t]\!] = [\![S]\!], \textsf{assume}\ c : [\![t]\!]_S}$$

$$\frac{k_j = (x_1\!:\!t_1) \to \cdots (x_n\!:\!t_n) \to \textsf{Type}}{\begin{array}{l}[\![S \\ \ \textsf{data}(T_1,p) : k_1\{\,|\,(c_{11} : t_{11}) \cdots |\,(c_{1n_1} : t_{1n_1})\} \\ \ \textsf{with} \cdots \\ \ \textsf{with data}(T_m,p) : k_m\{\,|\,(c_{m1} : t_{m1}) \cdots |\,(c_{mn_m} : t_{mn_m})\}]\!] \\ = [\![S]\!], \textsf{assume}\ T_1 : [\![k_1]\!]_S, \cdots, \textsf{assume}\ T_m : [\![k_m]\!]_S, \\ \qquad \textsf{assume}\ c_{11} : [\![t_{11}]\!]_S, \cdots \textsf{assume}\ c_{1n_1} : [\![t_{1n_1}]\!]_S, \\ \qquad \cdots, \\ \qquad \textsf{assume}\ c_{m1} : [\![t_{m1}]\!]_S, \cdots \textsf{assume}\ c_{mn_m} : [\![t_{mn_m}]\!]_S\end{array}}$$

Figure 5. Translation of AURA Signatures

If constructs $s\{e/x\}\ K$*, and* $K \in \{\textsf{Prop}, \textsf{Type}\}$*and* $S;E \vdash e : t$ *where* $t = \textsf{Type}, \textsf{Prop}$ *or* $S;E \vdash t : k$ *where* $k = \textsf{Type}, \textsf{Prop}$*, then constructs* $s\ K$

Proof. By induction on the structure of s. We only prove the case when $K = \textsf{Prop}$. The proof for the case when $K = \textsf{Type}$ is similar. The only possible cases are the following:

Case: s = **Prop**. trivial.

Case: s = x.
By assumption,

$$\textit{constructs}\ e\ \textsf{Prop} \qquad (1)$$
$$S;E \vdash e : t, S;E \vdash t : k \qquad (2)$$
$$\text{and } t = \textsf{Type}, \textsf{Prop} \text{ or } k = \textsf{Type}, \textsf{Prop} \qquad (3)$$

(1) contradicts with (2), (3)

Case: s = $(y\!:\!s_1) \to s_2$.
By assumption,

$$\textit{constructs}\ ((y\!:\!s_1) \to s_2)\{e/x\}\ \textsf{Prop} \qquad (1)$$

By inversion of (1),

$$\textit{constructs } s_2\{e/x\} \textsf{ Prop} \quad (2)$$

By I.H. on s_2,

$$\textit{constructs } s_2 \textsf{ Prop} \quad (3)$$

By definitions of *constructs* ,

$$\textit{constructs } ((y:s_1) \to s_2) \textsf{ Prop} \quad (4)$$

□

Lemma 24 (Translation Substitution)**.**
If $[\![e]\!]_{S;E_1,x:t,E_2} = e_1$, $S;E_1,x:t,E_2 \vdash e : t_1$, $S;E_1 \vdash u : t$, *and* $[\![u]\!]_{S;E_1} = u_1$, *and* $[\![e\{u/x\}]\!]_{S;E_1,E_2\{u/x\}} = e_2$ *, then* $e_2 = e_1\{u_1/x\}$.

Proof. By induction $\mathcal{E}$ on the typing derivation of e.

Case : $\mathcal{E}$ ends in VAR rule.
By Assumption,

$$e = x \quad (1)$$
$$[\![u]\!]_{S;E_1} = u_1 \quad (2)$$

By Translation rules,

$$e_1 = x \quad (3)$$

By Translation and Lemma 19 weakening,

$$[\![x\{u/x\}]\!]_{S;E_1,E_2\{u/x\}} = u_1 = e_1\{u_1/x\} \quad (4)$$

Case : $\mathcal{E}$ ends in VAR rule.
By Assumption,

$$e = y \neq x \quad (1)$$
$$[\![u]\!]_{S;E_1} = u_1 \quad (2)$$

By Translation rules,

$$e_1 = y \quad (3)$$

By Translation ,

$$[\![y\{u/x\}]\!]_{S;E_1,E_2\{u/x\}} = y = e_1\{u_1/x\} \quad (4)$$

Case : $\mathcal{E}$ ends in CTR rule.
By Assumption,

$$e = c \quad (1)$$
$$[\![u]\!]_{S;E_1} = u_1 \quad (2)$$

By Translation rules, and e_1 is closed,

$$[\![c\{u/x\}]\!]_{S;E_1,E_2\{u/x\}} = c = e_1\{u_1/x\} \quad (3)$$

Case: $\mathcal{E}$ ends in PF-RETURN rule.
By Assumption,

$$e = \textsf{return}_\textsf{p}\, p \quad (1)$$

$$\mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: S;E_1,x:t,E_2 \vdash p : P \\ \mathcal{E}_2 :: S;E_1,x:t,E_2 \vdash P : \textsf{Prop}\end{array}}{S;E_1,x:t,E_2 \vdash \textsf{return}_\textsf{p}\, p : \textsf{pf}\, P} \quad (2)$$

$$e_1 = \textsf{PF\_RET}\; Q\; q \quad (3)$$
$$[\![p]\!]_{S;E_1,x:t,E_2} = q \text{ and } [\![P]\!]_{S;E_1,x:t,E_2} = Q \quad (4)$$

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

$$[\![p\{u/x\}]\!]_{S;E_1,E_2\{u_1/x\}} = q\{u_1/x\} \quad (5)$$

By I.H. on $\mathcal{E}_2$,

$$[\![P\{u/x\}]\!]_{S;E_1,E_2\{u_1/x\}} = Q\{u_1/x\} \quad (6)$$

By Substitution,

$$S;E_1,E_2\{u_1/x\} \vdash p\{u/x\} : P\{u/x\} \quad (7)$$
$$S;E_1,E_2\{u_1/x\} \vdash P\{u/x\} : \textsf{Prop} \quad (8)$$

By Translation rule and (5)-(8),

$$[\![\textsf{return}_\textsf{p}\, p\{u/x\}]\!]_{S;E_1,E_2\{u_1/x\}} = (\textsf{PF\_RET}\; q\; Q)\{u_1/x\} \quad (9)$$

Case: $\mathcal{E}$ ends in ARR rule.
By Assumption,

$$e = (y:t_1) \to t_2 \quad (1)$$

$$\mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: S;E_1,x:t,E_2 \vdash t_1 : k_1 \\ \mathcal{E}_2 :: S;E_1,x:t,E_2,y:t_1 \vdash t_2 : k_2 \\ k_2 = \textsf{Prop}, \textsf{Kind} \\ k_1 = \textsf{Type}, \textsf{Prop} \text{ or } t_1 = \textsf{Type}, \textsf{Prop}\end{array}}{S;E_1,x:t,E_2 \vdash (y:t_1) \to t_2 : k_2} \quad (2)$$

$$e_1 = (y:s_1) \to s_2$$
where $s_1 = [\![t_1]\!]_{S;E_1,x:t,E_2}$
and $s_2 = [\![t_2]\!]_{S;E_1,x:t,E_2,y:t_1}$ (3)

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

$$[\![t_1\{u/x\}]\!]_{S;E_1,E_2\{u/x\}} = s_1\{u_1/x\} \quad (4)$$

By I.H. on $\mathcal{E}_2$,

$$[\![t_2\{u/x\}]\!]_{S;E_1,E_2\{u/x\},y:t_2\{u/x\}} = s_2\{u_1/x\} \quad (5)$$

By Translation rules,

$$[\![((y:t_1) \to t_2)\{u/x\}]\!]_{S,E_1,E_2\{u/x\}} = ((y:s_1) \to s_2)\{u_1/x\} \quad (6)$$

Case: $\mathcal{E}$ ends in APP rule.
By Assumption,

$$\mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: S;E_1,x:t,E_2 \vdash t_1 : (y:k_2) \to k_1 \\ \mathcal{E}_2 :: S;E_1,x:t,E_2 \vdash t_2 : k_2 \\ S;E_1,x:t,E_2 \vdash k_2 : u_2 \\ S;E_1,x:t,E_2 \vdash k_1\{t_2/y\} : u_1 \\ \ldots\end{array}}{S;E_1,x:t,E_2 \vdash t_1\, t_2 : k_1\{t_2/y\}} \quad (1)$$

$$e_1 = s_1\, s_2$$
where $s_1 = [\![t_1]\!]_{S;E_1,x:t,E_2}$
and $s_2 = [\![t_2]\!]_{S;E_1,x:t,E_2}$ (2)

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

$$[\![t_1\{u/x\}]\!]_{S;E_1,E_2\{u/x\}} = s_1\{u_1/x\} \quad (3)$$

By I.H. on $\mathcal{E}_2$,

$$[\![t_2\{u/x\}]\!]_{S;E_1,E_2\{u/x\}} = s_2\{u_1/x\} \quad (4)$$

By Translation rules,

$$[\![(t_1\, t_2)\{u/x\}]\!]_{S,E_1,E_2\{u/x\}} = (s_1\, s_2)\{u_1/x\} \quad (5)$$

□

We prove the following lemmas to show that if a term is a proof, a value, or it is a type or a kind, then its translation exists.

Lemma 25 (Term Translation Exists)**.**
If $S;E \vdash e : t$, $S \vdash \diamond$ *and one of the following holds:*

- $S;E \vdash t : \textsf{Prop}$
- *val*(e)
- $t = (x_1:t_1) \to \cdots (x_n:t_n) \to \textsf{Prop}$*, or* $t = (x_1:t_1) \to \cdots (x_n:t_n) \to \textsf{Type}$*, or* $t = \textsf{Kind}$

then exists w *such that* $[\![e]\!]_{S;E} = w$

Proof. By induction on the typing derivation of e. Most cases are immediate.

Case: $e = (x{:}t_1) \to t_2$

By assumption,

$S; E \vdash (x{:}t_1) \to t_2 : k_2$ (1)

By inversion of (1),

$S; E \vdash t_1 : k_1$ (2)

$S; E, x : t_1 \vdash t_2 : k_2$ (3)

$k_2 = \textsf{Kind}, \textsf{Type}, \textsf{Prop}$ (4)

$t_1 = \textsf{Type}, \textsf{Prop}$ or $k_1 = \textsf{Type}, \textsf{Prop}$ (5)

By I.H. on t_1,

$\exists u_1, s.t. [\![t_1]\!]_{S;E} = u_1$ (6)

By I.H. on t_2,

$\exists u_2, s.t. [\![t_2]\!]_{S;E,x:t_1} = u_2$ (7)

By translation rules, (6), (7),

$[\![(x{:}t_1) \to t_2]\!]_{S;E} = (x{:}u_1) \to u_2$ (8)

Case: $e = \lambda x{:}t_1.t_2$

By assumption,

$S; E \vdash \lambda x{:}t_1.t_2 : (x{:}t_1) \to k_2$ (1)

By inversion of (1),

$S; E \vdash t_1 : k_1$ (2)

$S; E, x : t_1 \vdash t_2 : k_2$ (3)

$S; E \vdash (x{:}t_1) \to k_2 : K$ (4)

$K = \textsf{Type}, \textsf{Prop}$ (5)

$t_1 = \textsf{Type}, \textsf{Prop}$ or $k_1 = \textsf{Type}, \textsf{Prop}$ (6)

when $K = \textsf{Type}$, by I.H. on (4)

$[\![(x{:}t_1) \to k_2]\!]_{S;E}$ exists (7)

By translation rules,

$[\![\lambda x{:}t_1.t_2]\!]_{=}\textsf{match ff return } [\![\lambda x{:}t_1.t_2]\!]_{S,E} \textsf{ with } \{\}$ (8)

when $K \neq \textsf{Type}$

By I.H. on (2), (6),

$\exists u_1, s.t. [\![t_1]\!]_{S;E} = u_1$ (9)

By inversion of (4),

$S; E, x : t_1 \vdash k_2 : K$ (10)

By I.H. on (3), (10),

$\exists u_2, s.t. [\![t_2]\!]_{S;E,x:t_1} = u_2$ (11)

By translation rules, (9), (11), (4),(5), and $K \neq \textsf{Type}$,

$[\![\lambda x{:}t_1.t_2]\!]_{S;E} = \lambda x{:}u_1.u_2$ (12)

Case: $e = e_1\, e_2$

By assumption,

$S; E \vdash e_1\, e_2 : t$ (1)

one of the following holds

I. $S; E \vdash t : \textsf{Prop}$

II. *val*$(e_1\, e_2)$

III. *constructs* t $\textsf{Prop}$, or *constructs* t $\textsf{Type}$

or $t = \textsf{Kind}$

By inversion of (1),

$S; E \vdash e_1 : (x{:}t_2) \to t_1$ (2)

$S; E \vdash e_2 : t_2$ (3)

$S; E \vdash t_2 : k_2$ (4)

$S; E \vdash t_1\{e_2/x\} : k_1$ (5)

val(e_2) or

$x \notin \mathit{fv}(t_1)$ and $k_1 = \textsf{Type}$ or $k_2 \in \{\textsf{Prop}, \textsf{Kind}\}$ (6)

$t = t_1\{e_2/x\}$ (7)

By Lemma Regularity, (2),

$S; E \vdash (x{:}t_2) \to t_1 : K$ (8)

By inversion of (8),

$S; E, x : t_2 \vdash t_1 : K$

and $K \in \{\textsf{Type}, \textsf{Prop}, \textsf{Kind}\}$ (9)

$t_2 \in \{\textsf{Type}, \textsf{Prop}\}$ or $k_2 \in \{\textsf{Type}, \textsf{Prop}\}$ (10)

I. $S; E \vdash t_1\{e_2/x\} : \textsf{Prop}$

By Substitution Lemma, (9), (6),

$S; E \vdash t_1\{e_2/x\} : K$

and $K \in \{\textsf{Type}, \textsf{Prop}, \textsf{Kind}\}$ (11)

since we assume that $S; E \vdash t_1\{e_2/x\} : \textsf{Prop}$

$S; E \vdash (x{:}t_2) \to t_1 : \textsf{Prop}$ (12)

By I.H. (2), (12),

$\exists s_1, [\![e_1]\!]_{S;E} = s_1$ (13)

When $t_2 \in \{\textsf{Type}, \textsf{Prop}\}$

By I.H. on (3),

$\exists s_2, [\![e_2]\!]_{S;E} = s_2$ (14)

By translation rule, $S; E \vdash t_1\{e_2/x\} : \textsf{Prop}$, (13), (14),

$[\![e_1\, e_2]\!]_{S;E} = s_1\, s_2$ (15)

When $k_2 \in \{\textsf{Type}, \textsf{Prop}\}$

By (6), and I.H. on (3),

$\exists s_2, [\![e_2]\!]_{S;E} = s_2$ (16)

By translation rule, $S; E \vdash t_1\{e_2/x\} : \textsf{Prop}$, (13), (16),

$[\![e_1\, e_2]\!]_{S;E} = s_1\, s_2$ (17)

II. *val*$(e_1\, e_2)$ and

By inversion of *val*$(e_1\, e_2)$,

val(e_1), *val*(e_2) (18)

By I.H. on (2), and (18),

$\exists s_1, [\![e_1]\!]_{S;E} = s_1$ (19)

By I.H. on (3), and (18),

$\exists s_2, [\![e_2]\!]_{S;E} = s_2$ (20)

By translation rules,

$[\![e_1\, e_2]\!]_{S;E} = s_1\, s_2$ (21)

III. *constructs* $t_1\{e_2/x\}$ $\textsf{Prop}$ or $t_1\{e_2/x\} = \textsf{Type}$

or $t_1\{e_2/x\} = \textsf{Kind}$

when $t_1\{e_2/x\} = \textsf{Kind}$

contradicts with (5) (22)

when *constructs* $t_1\{e_2/x\}$ $\textsf{Prop}$

By Lemma 23, (3), (4), (10),

constructs t_1 $\textsf{Prop}$ (23)

constructs $(x{:}t_2) \to t_1$ $\textsf{Prop}$ (24)

By I.H. on (2), (24),

$\exists s_1, [\![e_1]\!]_{S;E} = s_1$ (25)

when *constructs* $t_1\{e_2/x\}$ $\textsf{Type}$

By Lemma 23, (3), (4), (10),

constructs t_1 $\textsf{Type}$ (26)

constructs $(x{:}t_2) \to t_1$ $\textsf{Type}$ (27)

By I.H. on (2), (27),

$\exists s_1, [\![e_1]\!]_{S;E} = s_1$ (28)

By I.H. on (3), similar reasoning as in **I**,

$$\exists s_2, [\![e_2]\!]_{S;E} = s_2 \tag{29}$$

It is not the case that

$S;E \vdash t_1\{e_2/x\} : U,$ and $U = \textsf{Type}$

By translation rule,

$$[\![e_1\, e_2]\!]_{S;E} = s_1\, s_2 \tag{30}$$

□

Lemma 26 (Environment Translation Exists)**.**
If $S \vdash E$, $S \vdash \diamond$, and there is no equality binding in E, then exists Γ such that $[\![E]\!]_S = \Gamma$, and $\forall x \in dom(E)$, $[\![E(x)]\!]_{S;E} = \Gamma(x)$.

Proof (sketch): By induction on the structure of *E* using lemma 25. □

Lemma 27 (Signature Translation Exists)**.**

1. *If $\mathcal{E} :: S \vdash \mathsf{ok}$ then exists Σ such that $[\![S]\!] = \Sigma$, and $\forall c \in dom(S)$, $c \in dom(\Sigma)$ and $(\Sigma)(c) = [\![S(c)]\!]_{S;E}$.*
2. *If $\mathcal{E} :: S \vdash \diamond$ then exists Σ, Γ_0 such that $[\![S]\!] = \Sigma(\Gamma_0)$, and $\forall c \in dom(S)$, $c \in dom(\Sigma) \cup dom(\Gamma_0)$ and $(\Sigma, \Gamma_0)(c) = [\![S(c)]\!]_{S;E}$.*

Proof (sketch): By mutual induction on derivation $\mathcal{E}$ and use Lemma translation weakening (Lemma 20, Lemma 21), and Lemma term translation exists (Lemma 25). □

Lemma 28. *If $\mathcal{E} :: S; t_e; args; t_c; t_b; t_r \vdash \diamond$, $S;E \vdash t_e : \textsf{Prop}$,$S;E \vdash t_c : \textsf{Prop}$, $S;E \vdash t_r : \textsf{Prop}$, $S;E \vdash t_b : \textsf{Prop}$ and $[\![t_e]\!]_{S;E} = s_e$, $[\![t_c]\!]_{S;E} = s_c$, $[\![t_b]\!]_{S;E} = s_b$, $[\![t_r]\!]_{S;E} = s_r$, $[\![args]\!]_{S;E} = args'$, then $S; s_e; args'; s_c; s_b; s_r \vdash \diamond$*

Proof (sketch): By induction on derivation $\mathcal{E}$, use Translation Substitution 24. □

Lemma 29. *If $S; t_e; \cdot; t_c; t_b; P \vdash \diamond$, $c \in CtrsOf(T)$ and $c : \forall x_1 : t_1 \cdots \forall x_n : t_n \forall z_1 : s_1 \cdots \forall z_m : s_m.T x_1 \cdots x_n$, and $(\Sigma)[\Gamma, \Gamma_1] \vdash^{CC} c\, a_1 \cdots a_n\, y_1 \cdots y_k : t_c$, where $\{y_1, \cdots y_k\} = dom(\Gamma_1)$ and $y_i \notin fv(c\, a_1 \cdots a_n)$ then $t_b = \{t_c\}^P$*

Proof (sketch): By induction on $m - k$ □

Lemma 30. *If $S; T\, a_1\, \cdots a_n\, s_1 \cdots\, s_m; a_i \cdots, a_n; t_c; t_b; P \vdash \diamond$, $c \in CtrsOf(T)$ and $c : \forall x_1 : t_1 \cdots \forall x_n : t_n \forall z_1 : s_1 \cdots \forall z_m : s_m.T x_1 \cdots x_n$, and $(\Sigma)[\Gamma] \vdash^{CC} c\, a_1 \cdots a_{i-1} : t_c$, $(\Sigma)[\Gamma] \vdash^{CC} c\, a_1 \cdots a_n : t$ then $t_b = \{t\}^P$*

Proof (sketch): By induction on $n - i$, in the base case use Lemma 29. □

Next we show that well-typed AURA proofs are translated to well-typed CIC terms.

Lemma 31 (Correctness of Translation)**.**

1. *If $\mathcal{E} :: S \vdash \mathsf{ok}$, and $[\![S]\!] = \Sigma$ then $\mathcal{WF}(\Sigma^0; \Sigma)[\cdot]$.*
2. *If $\mathcal{E} :: S \vdash \diamond$, and $[\![S]\!] = \Sigma(\Gamma_0)$ then $\mathcal{WF}(\Sigma^0; \Sigma)[\Gamma_0]$.*
3. *If $\mathcal{E} :: S \vdash E$, $S \vdash \diamond$, and $[\![S]\!] = \Sigma(\Gamma_0)$, $[\![E]\!]_S = \Gamma_1$, then $\mathcal{WF}(\Sigma^0; \Sigma)[\Gamma_0, \Gamma_1]$.*
4. *If $\mathcal{E} :: S;E \vdash e : t$, $S \vdash \diamond$, and $[\![S]\!] = \Sigma(\Gamma_0)$, $[\![E]\!]_S = \Gamma_1$, $[\![e]\!]_{S;E} = e_1$ then $(\Sigma^0; \Sigma)[\Gamma_0, \Gamma_1] \vdash^{CC} e_1 : t_1$, and $[\![t]\!]_{S;E} = t_1$.*

Proof. By mutual induction on $\mathcal{E}$. We show the interest cases in proving (4).

Case: $e = x$

By assumption,

$$\frac{S \vdash E \quad E(x) = t}{S;E \vdash x : t} \tag{1}$$
$$[\![S]\!] = \Sigma(\Gamma_0) \tag{2}$$
$$[\![E]\!]_S = \Gamma_1 \tag{3}$$
$$[\![x]\!]_{S;E} = x \tag{4}$$

By 2.,

$$\mathcal{WF}(\Sigma^0; \Sigma)[\Gamma_0, \Gamma_1] \tag{5}$$

By Lemma 26,

$$\Gamma_1(x) = [\![t]\!]_{S;E} \tag{6}$$

By **Var** rule, (5), (6),

$$(\Sigma^0; \Sigma)[\Gamma_0, \Gamma_1] \vdash^{CC} x : [\![t]\!]_{S;E} \tag{7}$$

Case: e = c.

By assumption,

$$\frac{S \vdash E \quad S(c) = t}{S;E \vdash c : t} \tag{1}$$
$$[\![S]\!] = \Sigma(\Gamma_0) \tag{2}$$
$$[\![E]\!]_S = \Gamma_1 \tag{3}$$
$$[\![c]\!]_{S;E} = c \tag{4}$$

By 3.,

$$\mathcal{WF}(\Sigma^0; \Sigma)[\Gamma_0, \Gamma_1] \tag{5}$$

By Lemma 27,

$$(\Sigma, \Gamma_0)(c) = [\![S(c)]\!]_{S;E} \tag{6}$$

By Const rule, (5),(6),

$$(\Sigma^0, \Sigma)[\Gamma_0, \Gamma] \vdash^{CC} c : [\![S(c)]\!]_{S;E} \tag{7}$$

Case: $e = (x : t_1) \to t_2$

By assumption,

$$\frac{\begin{array}{l}\mathcal{E}_1 :: S;E \vdash t_1 : k_1 \quad \mathcal{E}_2 :: S;E, x : t_1 \vdash t_2 : k_2 \\ k_2 \in \{\textsf{Type}, \textsf{Prop}, \textsf{Kind}\} \\ t_1 \in \{\textsf{Type}, \textsf{Prop}\} \text{ or } k_1 \in \{\textsf{Type}, \textsf{Prop}\}\end{array}}{S;E \vdash (x : t_1) \to t_2 : k_2} \tag{1}$$
$$[\![S]\!] = \Sigma(\Gamma_0) \tag{2}$$
$$[\![E]\!]_S = \Gamma_1 \tag{3}$$
$$[\![(x : t_1) \to t_2]\!]_{S;E} = (x : s_1) \to s_2 \tag{4}$$

where $[\![t_1]\!]_{S;E} = s_1, [\![t_2]\!]_{S;E,x:t_1} = s_2$

By 3,

$$\mathcal{WF}(\Sigma^0; \Sigma)[\Gamma_0, \Gamma_1] \tag{5}$$

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

$(\Sigma^0; \Sigma)[\Gamma_0, \Gamma_1] \vdash^{CC} s_1 : K_1$

$$\text{where } K_1 = [\![k_1]\!]_{S;E} \tag{6}$$

By Well-formed environment rules,

$$S \vdash E, x : t_1 \tag{7}$$

By Environment translation rules,

$$[\![E, x : t_1]\!]_S = \Gamma_1, x : s_1 \tag{8}$$

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

$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1,x:s_1] \vdash^{CC} s_2 : K_2$
where $K_2 = [\![k_2]\!]_{S;E,x:t_1}$ (9)
By translation rules,
$K_2 \in \{\mathsf{PROP},\mathsf{SET},\mathsf{TYPE}\}$ (10)
$K_1 \in \{\mathsf{PROP},\mathsf{SET},\mathsf{TYPE}\}$ (11)
By **Ax** and **Prod** rule, (6), (9),
$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1] \vdash^{CC} (x:s_1) \to s_2 : K_2$ (12)

Case: $e = \lambda x\!:\!t_1.t_2$
By assumption,

$$\frac{\begin{array}{l}\mathcal{E}_1 :: S;E,x\!:\!t_1 \vdash t_2 : k_2\\ \mathcal{E}_2 :: S;E \vdash (x\!:\!t_1) \to k_2 : u\\ \mathcal{E}_3 :: S;E \vdash t_1 : k_1 \quad u \in \{\mathsf{Type},\mathsf{Prop}\}\\ k_1 \in \{\mathsf{Type},\mathsf{Prop}\} \text{ or } t_1 \in \{\mathsf{Type},\mathsf{Prop}\}\end{array}}{S;E \vdash \lambda x\!:\!t_1.t_2 : (x\!:\!t_1) \to k_2} \quad (1)$$

$[\![S]\!] = \Sigma(\Gamma_0)$ (2)
$[\![E]\!]_S = \Gamma_1$ (3)
$[\![\lambda x\!:\!t_1.t_2]\!]_{S;E} = e_1$ (4)
By 3.,
$\mathcal{WF}(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1]$ (5)
By inversion of (4), one of the following holds,
I. $u = \mathsf{Type}$
$[\![\lambda x\!:\!t_1.t_2]\!]_{S;E} =$
$\textsf{match}\ \mathsf{ff}\ \textsf{return}\ [\![(x\!:\!t_1) \to k_2]\!]_{S,E}\ \textsf{with}\ \{\}$
II. $[\![\lambda x\!:\!t_1.t_2]\!]_{S;E} = \lambda x\!:\!s_1.s_2$
where $[\![t_1]\!]_{S;E} = s_1, [\![t_2]\!]_{S;E,x:t_1} = s_2$, and $u = \mathsf{Prop}$
I.
By **match** rule,
$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1] \vdash^{CC} e_1 : [\![(x\!:\!t_1) \to k_2]\!]_{S,E}$ (6)
II.
By I.H. on $\mathcal{E}_3$,
$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1] \vdash^{CC} s_1 : K_1$
where $K_1 = [\![k_1]\!]_{S;E}$ (7)
By Well-formed environment rules,
$S \vdash E, x\!:\!t_1$ (8)
By Environment translation rules,
$[\![E,x:t_1]\!]_S = \Gamma_1, x:s_1$ (9)
By I.H. on $\mathcal{E}_1$, (8),
$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1,x:s_1] \vdash^{CC} s_2 : K_2$
where $K_2 = [\![k_2]\!]_{S;E,x:t_1}$ (10)
By translation rules, $\mathcal{E}_2$, $u = \mathsf{Prop}$, (10),
$[\![(x\!:\!t_1) \to k_2]\!]_{S;E} = (x\!:\!s_1) \to K_2$ (11)
By I.H. on $\mathcal{E}_2$, (11),
$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1] \vdash^{CC} (x\!:\!s_1) \to K_2 : \mathsf{PROP}$ (12)
By **Lam** rule, (7), (10), (12),
$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1] \vdash^{CC} \lambda x\!:\!s_1.s_2 : (x\!:\!s_1) \to K_2$ (13)

Case: $e = t_1\, t_2$
By assumption,

$$\frac{\begin{array}{l}\mathcal{E}_1 :: S;E \vdash t_1 : (x\!:\!k_2) \to k_1\\ \mathcal{E}_2 :: S;E \vdash t_2 : k_2\\ \mathcal{E}_3 :: S;E \vdash k_2 : u_2\\ \mathcal{E}_4 :: S;E \vdash k_1\{t_2/x\} : u_1\\ \mathit{val}(t_2)\\ \text{or } (x \notin \mathit{fv}(k_1)\\ \quad \text{and } (u_1 = \mathsf{Type} \text{ or } u_2 \in \{\mathsf{Prop},\mathsf{Kind}\}))\end{array}}{S;E \vdash t_1\, t_2 : k_1\{t_2/x\}} \quad (1)$$

$[\![S]\!] = \Sigma(\Gamma_0)$ (2)
$[\![E]\!]_S = \Gamma_1$ (3)
$[\![t_1\, t_2]\!]_{S;E} = s_1\, s_2$ (4)
where $[\![t_1]\!]_{S;E} = s_1, [\![t_2]\!]_{S;E} = s_2$
By 3.,
$\mathcal{WF}(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1]$ (5)
By I.H. on $\mathcal{E}_1$,
$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1] \vdash^{CC} s_1 : [\![(x\!:\!k_2) \to k_1]\!]_{S;E}$ (6)
By I.H. on $\mathcal{E}_2$,
$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1] \vdash^{CC} s_2 : [\![k_2]\!]_{S;E}$ (7)
By translation rules,
$[\![(x\!:\!k_2) \to k_1]\!]_{S;E} = (x\!:\!w_2) \to w_1$ (8)
where $[\![k_2]\!]_{S;E} = w_2, [\![k_1]\!]_{S;E,x:k_2} = w_1$
By **App** rule, (6), (7), (8),
$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1] \vdash^{CC} s_1\, s_2 : w_1\{s_2/x\}$ (9)
By Lemma Translation Substitution 24, (8),
$[\![k_1\{t_2/x\}]\!]_{S;E} = w_1\{s_2/x\}$ (10)

Case :$e = \textsf{match}\ e\ t\ \textsf{with}\ \{\,|\,b_1 \cdots |\,b_n\}$
By assumption,

$$\frac{\begin{array}{l}\mathcal{E}_0 :: S;E \vdash e : s \quad s = \mathit{ctr}\ a_1\ a_2 \cdots a_n\\ S(\mathit{ctr}) = (x_1 : t_1) \to \cdots (x_n : t_n) \to u\\ \mathit{branches\_cover}\ S\ \mathit{branches}\ \mathit{ctr}\\ \mathcal{E}_1 :: S;E;s;(a_1,\cdots,a_n) \vdash b_1 \cdots, b_n : t\\ \mathcal{E}_2 :: S;E \vdash s : u \quad \mathcal{E}_3 :: S;E \vdash t : u\\ u \in \{\mathsf{Type},\mathsf{Prop}\}\end{array}}{S;E \vdash \textsf{match}\ e\ t\ \textsf{with}\ \{\,|\,b_1 \cdots |\,b_n\} : t} \quad (1)$$

$[\![S]\!] = \Sigma(\Gamma_0)$ (2)
$[\![E]\!]_S = \Gamma_1$ (3)
$[\![\textsf{match}\ e\ t\ \textsf{with}\ \{\,|\,b_1 \cdots |\,b_n\}]\!]_{S;E} = e_1$ (4)
By 3.,
$\mathcal{WF}(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1]$ (5)
By inversion of (4),
$\textsf{match}\ [\![e]\!]_{S;E}\ \mathsf{return}\ [\![t]\!]_{S;E}\ \textsf{with}\ \{\,|\,[\![b_1]\!]_{S;E} \cdots |\,[\![b_n]\!]_{S;E}\}$
and $u = \mathsf{Prop}$ (6)
By I.H. on $\mathcal{E}_0$,
$(\Sigma^0;\Sigma)[\Gamma_0,\Gamma_1] \vdash^{CC} [\![e]\!]_{S;E} : [\![s]\!]_{S;E}$ (7)
By translation rules, and $S;E \vdash s : \mathsf{Prop}$,
$[\![s]\!]_{S;E} = \mathit{ctr}\ [\![a_1]\!]_{S;E} \cdots [\![a_n]\!]_{S;E}$ (8)
By Lemma 27,
$\Sigma(\Gamma_0)(\mathit{ctr}) =$
$[\![(x_1 : t_1) \to \cdots (x_n : t_n) \to u]\!]_{S;E}$ (9)
By translation rules, and $u = \mathsf{Prop}$,
$\Sigma(\Gamma_0)(\mathit{ctr}) =$
$(x_1 : [\![t_1]\!]_{S;E}) \to \cdots (x_n : [\![t_n]\!]_{S;E}) \to \mathsf{PROP}$ (10)
By I.H. on $\mathcal{E}_2$,

$$(\Sigma^0; \Sigma)[\Gamma_0, \Gamma_1] \vdash^{CC} [\![s]\!]_{S;E} : \mathsf{PROP} \quad (11)$$

By I.H. on $\mathcal{E}_3$,

$$(\Sigma^0; \Sigma)[\Gamma_0, \Gamma_1] \vdash^{CC} [\![t]\!]_{S;E} : \mathsf{PROP} \quad (12)$$

By inversion on $\mathcal{E}_1$,

$$\text{constructors of ctr} = \{c_1 \cdots c_n\}, S(c_i) = tc_i \quad (13)$$

$$\forall i, b_i = c_i \ \rightarrow f_i \text{ and } S; E \vdash f_i : tb_i \quad (14)$$

$$\forall i, S; s; (a_1, \cdots a_n); tc_i; tb_i; t \vdash \diamond \quad (15)$$

By I.H. on (14),

$$(\Sigma^0; \Sigma)[\Gamma_0, \Gamma_1] \vdash^{CC} [\![f_i]\!]_{S;E} : [\![tb_i]\!]_{S;E} \quad (16)$$

By Lemma 27, (13),

$$\Sigma(\Gamma_0)(c_i) = [\![tc_i]\!]_{S;E} \quad (17)$$

By Lemma 28, (14),

$$\begin{array}{l} S; [\![s]\!]_{S;E}; ([\![a_1]\!]_{S;E}, \cdots [\![a_n]\!]_{S;E}); \\ \quad [\![tc_i]\!]_{S;E}; [\![tb_i]\!]_{S;E}; [\![t]\!]_{S;E} \end{array} \vdash \diamond \quad (18)$$

By Lemma 30, (8), (16),(17), (18),

$$\begin{array}{l} (\Sigma^0; \Sigma)(\Gamma_0, \Gamma_1) \vdash^{CC} [\![f_i]\!]_{S;E} \\ \qquad : \{c_i\ [\![a_1]\!]_{S;E}, \cdots [\![a_n]\!]_{S;E}\}^{[\![t]\!]_{S;E}} \end{array} \quad (19)$$

By **match** rule, (7), (8), (19),

$$\begin{array}{l} (\Sigma^0; \Sigma)(\Gamma_0, \Gamma_1) \vdash^{CC} \ \mathbf{match}\ [\![e]\!]_{S;E} \mathsf{return}\ [\![t]\!]_{S;E} \\ \qquad \mathbf{with}\ \{\mid [\![b_1]\!]_{S;E} \cdots \mid [\![b_n]\!]_{S;E}\} \\ \qquad : [\![t]\!]_{S;E} \end{array} \quad (20)$$

□

We need to build a connection between reductions of AURA proofs and the reduction of the CIC terms that are the translation images. The only special reduction rule in AURA is the commute rule (BINDC), which correspond to the following reduction rule for CIC terms.

Special Reduction Rule:

$(\lambda x\!:\!t.t_1)((\lambda y\!:\!s.t_2)u) \rightarrow_{\beta'} (\lambda y\!:\!s.((\lambda x\!:\!t.t_1)t_2))u$

We prove that CIC augmented with β' is also strongly normalizing. We use $\mathbf{SN}(\beta)$ to denote the set of terms that are strongly normalizing under β reductions in CIC; similarly, $\mathbf{SN}(\beta\beta')$ is the set of terms that are strongly normalizing under the β and β' reduction rules.

Lemma 32 (Strong normalization of $\beta\beta'$-reduction in CIC)**.** *For all term $t \in \mathbf{SN}(\beta)$, $t \in \mathbf{SN}(\beta\beta')$.*

Proof. We use the technique presented in Lindley's thesis [20]. 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 $t \rightarrow_{\beta'} t'$ then $\beta(t') \leq \beta(t)$, 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 $t \rightarrow_{\beta'} t'$, $(\beta(t'), \delta(t')) < (\beta(t), \delta(t))$. Consequently, for all term $t \in \mathbf{SN}(\beta)$, $t \in \mathbf{SN}(\beta\beta')$. □

Now we prove that the reductions in CIC augmented with the β' reduction rule simulates the reduction in the prop fragment of AURA.

Lemma 33 (Simulation)**.** *If $S; E \vdash t : k$, and $\lceil t \rceil_{S;E} \Longrightarrow \bar{t}'$, and $[\![t]\!]_S = u$, $[\![t']\!]_S = u'$, then $u \rightarrow^{+}_{\beta,\beta'} u'$.*

Proof (sketch): By induction on the typing derivation $S; E \vdash t : k$. Notice that there is no proof reduction rules for $\boxed{\lambda x\!:\!t_1.t_2}$, which is translated to a stuck pattern-matching term in CIC. □

Theorem 34 (Strong Normalization)**.** *If $S; \cdot \vdash e : P$ and $S; \cdot \vdash P :$ **Prop** then e is strongly normalizing.*

Proof. By Lemma 25, $[\![e]\!]_S$ exists. By Lemma 33, and Lemma 32. A diverging path in AURA's prop fragment implies a diverging path in CIC. Since CIC is strongly normalizing, AURA's prop fragment is also strongly normalizing. □