AURA: Preliminary Technical Results

University of Pennsylvania
Technical Report MS-CIS-08-10

April 17, 2008

Limin Jia Jeffrey A. Vaughan Karl Mazurak Jianzhou Zhao Luke Zarko Joseph Schorr
Steve Zdancewic

University of Pennsylvania
{liminjia, vaughan2, mazurak, jianzhou, zarko, jschorr, stevez}@seas.upenn.edu

Abstract

This paper presents AURA, a programming language for access control that treats ordinary programming constructs (e.g., integers and recursive functions) and authorization logic constructs (e.g., principals and access control policies) in a uniform way. AURA is based on polymorphic DCC and uses dependent types to permit assertions that refer directly to AURA values while keeping computation out of the assertion level to ensure tractability. The main technical results of this paper include fully mechanically verified proofs of the decidability and soundness for AURA's type system, and a prototype typechecker and interpreter.

1. Introduction

There can be no universal definition of security. Every piece of confidential data and every sensitive resource may have specialized access control requirements. At the same time, almost every modern computer system stores some private information or provides a service intended only for certain clients. To ensure that only allowed principals—human users or other computer systems—can reach the protected resources, these access control requirements must be carefully defined and enforced. An *authorization policy* specifies whether a request by a principal to access a resource should be granted, and a *reference monitor* mediates all access to the resource, ensuring that the handling of requests complies with the authorization policy.

One significant challenge in building secure systems that enforce access control is that, as the number of resources and principals grows, specifying the authorization policy becomes more difficult. The situation is further complicated in decentralized or distributed settings, where resources may have different owners and the principals may have non-trivial trust relationships. Once the policies become sufficiently complex, understanding which principals may access which resources is itself a daunting problem. Consequently, reference monitors that enforce such policies also become complex, which is not a desired situation when (as in a conventional access control scheme) the reference monitor is part of the trusted computing base.

To help mitigate this complexity, researchers have proposed *authorization logics* that facilitate reasoning about principals, requests, and policy assertions [5, 14, 20, 1, 2]. Several of these logics have been concerned with specifying access control policies in distributed settings [44, 6, 11, 21, 20]. Part of the appeal of authorization logics is that proofs of propositions in the logic can act as *capabilities* that provide the reference monitor with evidence that a given request should be granted. As proposed by Appel and Felten [6], *proof-carrying authorization* places the burden of validating the authorization decision on the principal requesting access. Moreover, the explicit proofs can be logged for future auditing, which can help track down bugs in the authorization policy [39].

Authorization logics are rich and concise languages for specifying access control policies, abstracting away low-level details like authentication and cryptography. Unfortunately, these logics are rather removed from the languages used to write software that must respect the access control policies; tools like typecheckers that help the programmer write correct programs will not necessarily help the programmer make correct use of an authorization logic. This is especially problematic in the case of a reference monitor, which has the task of enforcing policies written in an authorization logic and must be considered part of the trusted computing base.

This paper presents the design of AURA, a domain-specific programming language that incorporates a constructive authorization logic based on DCC [4, 2] as part of its type system. Rather than mediating between programs and policy statements written in two distinct languages, AURA uses *dependent types* to permit policy statements that refer directly to AURA values (like integers or datatype constructors). 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 principal p is permitted to play s:

$$(s\,{:}\,\mathit{Song}) \rightarrow (p\,{:}\,\mathsf{prin}) \rightarrow \mathsf{pf}\ (\mathsf{self\ says}\ \mathit{MayPlay}\ p\ s) \rightarrow \mathit{Unit}.$$

As indicated by this type, AURA programs may construct and manipulate authorization proofs just as they might other program values, and the AURA programming model provides notions of principals (p), authority (self), and policy assertions (*MayPlay*) in addition to standard functional language features like higher-order functions, polymorphism, and recursive algebraic datatypes. In addition, security-relevant implementation details—like the creation of audit trails or the cryptographic interpretation of certain logical statements—can be handled automatically with little to no programmer intervention.

Because policy assertions are part of AURA's type system, deciding whether to grant access amounts to typechecking a proof object. This can be performed by AURA's runtime, removing individual reference monitors from the trusted computing base. Moreover, any program written in AURA benefits from the immediate availability of the authorization logic; many misbehaving programs can now be ruled out at compile time. Finally, DCC, on which AURA is based, has been shown to be useful in representing other forms of language-based security, such as the type-based enforcement of information-flow properties as found in Jif [30] or FlowCaml [34]; AURA thus represents a promising avenue for further work in connecting these concepts.

The main contributions of this paper are as follows:

- We present the design of core AURA, a language with support for first-class, dependent authorization policies.
- We give a fully machine-checked proof of type soundness for the core language.
- We also give a fully machine-checked proof of decidability of typechecking for AURA.
- We describe a prototype implementation of a typechecker and interpreter and give sample programs.

Typical dependently typed languages (see Section 6) use types to encode precise program specifications. Our goal is different; AURA uses dependent types to naturally connect data with proofs for run-time policy enforcement. Compared with a conventional dependently typed language, AURA adds some features—assertion types, digitally signed objects as proofs, the says and pf modalities—and restricts or removes others—only values may appear in dependent types. The result is a system tuned for dynamic authorization but unsuitable for, e.g., static program verification.

Our proof of soundness is implemented in Coq and encompasses all of AURA's features, including higher order and polymorphic types, mutually recursive datatypes and propositions, a restricted form of dependent types, and authorization proofs. We believe that the mechanized proof is of independent value, as parts of the proof may be reused in other settings.

The rest of this paper focuses on the novel core features of AURA. The next section introduces AURA's programming model and illustrates its novel features by example. Section 3 gives a formal account of the core AURA language, including its type system and operational semantics, along with our main technical results, soundness and decidability of typechecking. Section 4 describes our prototype implementation. Section 5 gives a larger scale example demonstrating how AURA's features work in concert. Section 6 situates AURA with respect to related work, especially prior work on authorization logics and languages with dependent types. Finally, Section 7 concludes with a discussion of future avenues for extending AURA.

AURA as we present it is most suitable as a compilation target for a more convenient surface syntax. As such, we defer the important (and practical) issues of type inference, pattern match compilation, and the like to future work. Additional topics for future study include authentication, credential revocation, the interpretation of AURA values in cryptography, and integration with mixed language (e.g. C or .NET) systems.

2. Programming in AURA

AURA is intended to be used to implement reference monitors [12] for access control in security sensitive settings. A reference monitor mediates access by allowing and denying requests to a resource (based, in this case, on policy specified in an authorization logic). It must also log accesses to enable *ex post facto* audit. This latter point we have covered in detail elsewhere [39] (although we discuss logging briefly in Section 2.3); in this paper we concentrate on the details of integrating general purpose programming with an authorization logic.

The potential design space of dependently-typed languages is quite large, and there are many challenges in striking a good balance between expressiveness and tractability of typechecking. AURA's design strives for simplicity, even at the cost of expressiveness. This section describes AURA's design, concentrating on the features relevant to access control.

As alluded to by the function *playFor* in the introduction, we use an AURA implementation of a jukebox server as a running example throughout this paper. The full example is given in Section 5; the rest of this section will illustrate *playFor* in more detail.

2.1 AURA as an authorization logic

We first turn our attention to AURA's assertions, which are based on the polymorphic core calculus of dependency (DCC) [4] and in particular on DCC's interpretation as an authorization logic [2].[1] In both DCC and AURA, an indexed monad says associates propositions with principals. The statement a says P holds when a proof for P is known, when a says P logically follows from monad operations that we will describe shortly, or when the principal a directly affirms P. It is critical to note, however, that a says P does not imply P. We augment DCC with dependent types, which allow principals to assert propositions about data, and with the constructs say and sign, which allow for the aforementioned direct affirmations.

Principals in AURA, written a, b, etc. and having type prin, represent distinct components of a software system. They may correspond to human users or system components such as an operating system kernel, a particular server, and so on. Formally, principals are treated as special values in AURA; they are characterized by their ability to index the family of says monads.

As 'a says' is a *monad* [41], we can construct a term of type a says P from a proof p of P using the operation return $a\ p$. A proof encapsulated in a says monad cannot be used directly; rather, the monad's bind operation, written $(\mathsf{bind}\ p\ (\lambda x{:}P.\ q))$ allows x to stand in for the proof encapsulated by p and appear in the expression q.

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

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

assert that a gives herself permission to play *freebird* and b delegates to a the authority to make any variety of *MayPlay* statement on his behalf. These two terms may be used to create a proof of b says (*MayPlay a freebird*) as follows:

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

Such a proof might have direct utility—it could be passed to the *playFor* function if self is b—or it might become part of a larger chain of reasoning.

In addition to uses of return, 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 $\mathsf{sign}(a, P)$ has type a says P; intuitively we may think of it as a digital signature using a's private key on proposition P. Such a value is intended to have a stable meaning as it is passed throughout a distributed system.

[1] AURA is most similar to the cut-down variant of DCC called CDD [3]. Full DDC, as opposed to AURA and CDD, features a relaxed typing rule for bind and admits type coercions unsuitable for access control.

Only the principal *a*—or, equivalently, programs with access to *a*'s private key—should be able to create a term of the form $\mathsf{sign}(a, P)$. 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 $\mathsf{sign}(\mathsf{self}, P)$, where self is a distinguished principal representing the current runtime authority.

It is worth noting that a principal can assert any proposition, even *False*. Because assertions are confined to the monad—thanks to the non-interference property of DCC—such an assertion can do little harm apart from making that particular principal's own assertions inconsistent. In practice, it is useful to restrict the kinds of assertions that various principals can make, but, *a priori*, AURA requires no such constraints.

The concept of a program's runtime authority already has a natural analog in the operating system world—a UNIX process, for example, has an associated user ID that often, but not always, corresponds to the user who started the process. In a more distributed setting, running under the authority of a can indeed be represented by possession of a's private key. In such a setting objects of the form $\mathsf{sign}(a, P)$ can be represented by actual digital signatures, and principal identifiers—which, in AURA, are first class values of type prin—can be thought of as public keys.

The restriction of authority to a single principal is only for simplicity's sake; although syntax would need to be changed, nothing in our development would conflict with a more complex notion of authority. AURA currently provides no means of *transferring* authority, in effect disallowing programs from directly manipulating private keys; this prevents AURA programs from creating new principals (i.e., key pairs) at runtime but also trivially disallows the accidental disclosure of private keys. Were AURA to be extended with support for dynamically generated principals, the addition of information flow tracking could assist in ensuring that private keys stay sufficiently private.

2.2 Authorization proofs and dependent types

By defining assertions as types and proofs as terms we are taking advantage of the well-known Curry-Howard Isomorphism [19, 24] between logic and programming languages. One benefit to this approach is that AURA programs use the same constructs to manipulate both data and proofs. More critically it provides—via dependent typing, which allows types to mention terms—an elegant way for access control statements to mention data. For instance, in the example given earlier, *freebird* is data that appears at the assertion (i.e. type) level. The type signature for the function *playFor* in the introduction is another example of such dependency.

AURA incorporates dependent types directly—in contrast to, for example, using GADTs [33] or static equality proofs [36] to simulate the required dependencies. Such an approach allows straightforward use of data at the type level and avoids replication of the same constructs in both static and dynamic form, but unconstrained use of dependent types can quickly lead to an undecidable typing judgment. Moreover, care must be taken to separate effectful computations from pure proof objects.

Much like CIC [18], AURA has separate universes Type and Prop, with the constants Type and Prop themselves being classified by Kind. The previously mentioned assertion *MayPlay*, for instance, would be given the assertion type prin → *Song* → Prop. Unlike CIC, both types of kind Type and propositions of kind Prop describe data that may be available at runtime. Propositions, however, are required to be completely computation-free: propositions never reduce and AURA does not employ type-level reduction during typechecking, meaning that only dependencies on values (i.e., well-formed normal forms) for which equality comparison is available can be used in non-trivial ways. This turns out to be enough to ensure the decidability of AURA's type system.

AURA offers a type-refining equality test on *atomic* values—for instance, principals and booleans—as well as a dynamic cast between objects of equivalent types, which prove necessary for certain equalities that arise only at runtime. For example, when typechecking $\mathsf{if}\ \mathsf{self} = a\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, the fact that $\mathsf{self} = a$ is automatically made available while typechecking e_1 (due to the fact that prin is an atomic type), and hence proofs of type $\mathsf{self}\ \mathsf{says}\ P$ can be cast to type $a\ \mathsf{says}\ P$ and vice-versa.

The distinction between Type and Prop is also illustrated by the previously introduced say and sign. On the one hand, say P certainly belongs in Type's universe. We intend it to be reduced by our operational semantics—and this reduction is an effectful (if trivial) computation dependent on a program's runtime authority. On the other hand, $\mathsf{sign}(a, P)$ should be of type $a\ \mathsf{says}\ P$, which, like P, is of kind Prop. To solve this dilemma we introduce the modality pf : Prop → Type, allowing us to give say P the type pf $(\mathsf{self}\ \mathsf{says}\ P)$ of kind Type. The pf modality comes equipped with its own bind and return operations, allowing proofs to be manipulated by computations while keeping the worlds of computations and assertions separate.

AURA's dependent types also address something that might have seemed odd about our cryptographic interpretation of the says monad, namely that one most often thinks of digitally signing *data*, whereas $\mathsf{sign}(a, P)$ signs only an assertion. With dependent types, however, this issue evaporates, as an assertion can refer to whatever data might be endorsed. We find this design compelling, because a digital signature on raw data does not necessarily have a sensible meaning; signing only propositions ensures that the signed data is attributed with some semantics, just as, for example, a physical signature on a contract will indicate whether the signer is party to the contract or merely a witness.

Our previous work [39] addressed only the Prop fragment of AURA; it did not consider the Type level constructs necessary for programming. This paper presents a complete programming model that includes recursive datatypes, constructs such as the pf modality and say operator, and dynamic type refinement in if statements, which are absent from our previous work. The new additions to the language (particularly recursive datatypes) significantly complicate the metatheory.

2.3 Auditing in AURA

Passing proofs at runtime is also useful for after the fact auditing of AURA programs. The full details are given elsewhere [39] but we note that, when full proofs are logged for every resource access, it becomes possible to determine *how* access was granted at a very fine granularity. This is of great importance when the intent of some institutional policy is not properly reflected in the actual rules enforced by a software system—for example, an auditor can examine the proof that allowed an unwanted access to take place and determine whether and where authority was improperly delegated.

These guarantees can be made as long as the interface to the resources of interest is sufficiently rich: we can simply decree that every interface function—that is, a function that wraps a lower level operating system call—must write its arguments to the log. There are no constraints on what the rest of the reference monitor may do other than that it must respect this interface; it is not possible to inadvertently add a path through the program that causes insufficient information to be logged. This is in keeping with AURA's general philosophy of resilience toward programmer mistakes.

Returning to *playFor*, let us assume that there exists a native function *rawPlayFor* : *Song* → *Unit* that is not security-aware and hence is not available to the programmer. We define the interface function *playFor* as simply

$$\lambda s\!:\!Song.\ \lambda p\!:\!\mathsf{prin}.\ \lambda proof\!:\!\mathsf{pf}\ (\mathsf{self\ says}\ MayPlay\ p\ s).\ rawPlayFor\ s.$$

Because *playFor* is an interface function—i.e., because it has access to *rawPlayFor*—its arguments will automatically be logged, and because the access control policy is entirely encoded in *playFor*'s signature, the log will automatically contain everything an auditor needs to determine precisely how any song was authorized to be played.

3. The AURA Core Language

This section presents the main technical contributions of this paper, namely a formal description of the AURA core language, its type system, operational semantics, and the corresponding proofs of type soundness and decidability of typechecking.

We adopt the design philosophy of elaboration-style semantics (as used, for example, by Lee *et. al* [28]): the AURA intermediate language is intended to make typechecking as explicit as possible. Following this principle, our design eschews complex pattern matches, equality tests over complex values, and implicit casts. Our goal was to cleanly divide the compiler into two parts: an elaboration phase that uses inference, possibly with heuristics and programmer-supplied hints, to construct an internal representation that makes all type information explicit; and a compilation phase that processes the fully elaborated intermediate representation into executable code.

3.1 AURA core syntax

As described above, AURA is a call-by-value polymorphic lambda calculus. It consists of a "term-level" programming language (whose expressions are classified by types of kind Type) for writing algorithms and manipulating data and a "proof-level" assertion language (whose expressions are classified by propositions of kind Prop) for writing proofs of access control statements. These two languages share many features (λ-abstraction, application, constructors, etc.) and, because of dependent types, propositions and types may both mention terms of either sort. To simplify the presentation of AURA, it makes sense to unify as many of these constructs as possible. We thus adopt a lambda-cube style presentation [10] that uses the same syntactic constructs for terms, proofs, types, and propositions. Different categories are distinguished by the type system as necessary. This approach also has the appeal of greatly reducing the number of objects in the language, which simplifies both the metatheory and implementation. Our design was significantly influenced by the Henk intermediate language [25], which also adopts this compact representation.

The lambda-cube terms of the AURA core syntax are given by:

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

Here, x ranges over variables, and *ctr* ranges over programmer-defined constructors created using datatype declarations as described below. In addition to the standard lambda abstraction, application, and dependent arrows, AURA also has a pattern matching construct and an explicit typecast. In the expression $\mathsf{match}\ t_1\ t_2\ \mathsf{with}\ \{b\}$, t_1 is the term under analysis, t_2 is the return type, and *b* is a list of branches that t_1 is matched against. Type annotation t_2 ensures that typechecking is straightforward even when the set of branches is empty. The explicit cast $\langle t_1 : t_2 \rangle$ ensures (safely) that t_1 be considered to have type t_2.

To express and reason about access control, AURA extends the core syntax above with additional terms. Here, and throughout the rest of the paper, we use metavariable conventions that make it easier to recall constraints placed on a term by the type system: a ranges over principals, P ranges over propositions, p ranges over proofs, e ranges over program expressions, and v stands for values. All of these metavariables are synonymous with t, which we use to indicate syntactic objects of any flavor. The AURA-specific syntax is given by[2]:

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

3.2 Typechecking AURA

AURA's type system contains the following judgments:

Well-formed signature	$S \vdash \diamond$
Well-formed typing environment	$S \vdash E$
Well-typed term	$S; E \vdash t : s$
Well-typed match branches	$S; E; s; args \vdash branches : t$

Figure 1 shows the term typechecking rules. We omit the rules for typechecking signatures and branches, though we describe their salient features below. The full type system can be found in the Appendices and the Coq implementation.

In all these judgments, *S* is a signature that declares types, propositions, and assertions (described in more detail below). A typing environment *E* maps variables to their types as usual, but it also records the hypothetical equalities among atomic run-time values. In the definition of environments below, a binding $x \sim (v_1 = v_2){:}t$ indicates that values v_1 and v_2 have type t, and that the run-time values of v_1 and v_2 are equal.

$$\text{Environments}\quad E \ ::= \ \cdot \mid E, x\!:\!t \mid E, x \!\sim\! (v_1 = v_2){:}t$$

3.3 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 together these data definitions and, as a consequence, a well-formed signature can be thought of as map from constructor identifiers to their types. We present the formal grammar and typing rules for signatures, which are largely straightforward, in Appendix A; here we explain signatures via examples.

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

```
data List :Type → Type {
|nil :(t:Type) → List t
|cons :(t:Type) → t → List t → List t
}
```

AURA's type system rules out data declarations that require nontrivial equality constraints at the type level. For example, the following GADT-like declaration is ruled out, since *Bad t u* would imply $t = u$:

```
data Bad :Type → Type → Type {
|bad :(t:Type) → Bad t t
}
```

Logical connectives like conjunction and disjunction can be encoded using dependent propositions, as in Coq and other type-based provers. For example:

[2] In the Coq development, these constructs are represented using constants and term application.

$$\frac{S \vdash E}{S;E \vdash \mathsf{Type} : \mathsf{Kind}}\ \textsc{Wf-tm-type} \qquad \frac{S \vdash E}{S;E \vdash \mathsf{Prop} : \mathsf{Kind}}\ \textsc{Wf-tm-prop}$$

$$\frac{S \vdash E \quad S(\mathit{ctr}) = t}{S;E \vdash \mathit{ctr} : t}\ \textsc{Wf-tm-ctr} \qquad \frac{S \vdash E \quad E(x) = t}{S;E \vdash x : t}\ \textsc{Wf-tm-fv} \qquad \frac{S;E, x\!:\!t_1 \vdash t_2 : k_2 \quad k_2 \in \{\mathsf{Type}, \mathsf{Prop}, \mathsf{Kind}\}}{S;E \vdash (x\!:\!t_1) \to t_2 : k_2}\ \textsc{Wf-tm-arr}$$

$$\frac{S;E \vdash t : k \quad S;E, x\!:\!t \vdash u : k_1 \quad S;E \vdash (x\!:\!u) \to k_1 : k_2 \quad k \in \{\mathsf{Type}, \mathsf{Prop}, \mathsf{Kind}\} \quad k_2 \in \{\mathsf{Type}, \mathsf{Prop}\}}{S;E \vdash \lambda x\!:\!t.\ u : (x\!:\!t) \to k_1}\ \textsc{Wf-tm-abs}$$

$$\frac{S;E \vdash t_1 : (x\!:\!u_2) \to u \quad S;E \vdash t_2 : u_2 \quad \mathit{val}(t_2) \text{ or } x \notin \mathit{fv}(u)}{S;E \vdash t_1\, t_2 : \{x/t_2\}u}\ \textsc{Wf-tm-app}$$

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

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

$$\frac{S;E \vdash a : \mathsf{prin} \quad S;E \vdash P : \mathsf{Prop}}{S;E \vdash a\ \mathsf{says}\ P : \mathsf{Prop}}\ \textsc{Wf-tm-says} \qquad \frac{S;E \vdash a : \mathsf{prin} \quad \mathit{val}(a) \quad S;E \vdash p : P \quad S;E \vdash P : \mathsf{Prop}}{S;E \vdash \mathsf{return}_s\ a\ p : a\ \mathsf{says}\ P}\ \textsc{Wf-tm-says-ret}$$

$$\frac{S;E \vdash e_1 : a\ \mathsf{says}\ P \quad S;E \vdash e_2 : (x\!:\!P) \to a\ \mathsf{says}\ Q \quad x \notin \mathit{fv}(Q)}{S;E \vdash \mathsf{bind}_s\ e_1\ e_2 : a\ \mathsf{says}\ Q}\ \textsc{Wf-tm-says-bind}$$

$$\frac{S;\cdot \vdash a : \mathsf{prin} \quad S;\cdot \vdash P : \mathsf{Prop}}{S;E \vdash \mathsf{sign}(a, P) : a\ \mathsf{says}\ P}\ \textsc{Wf-tm-sign} \qquad \frac{S;E \vdash P : \mathsf{Prop}}{S;E \vdash \mathsf{say}\ P : \mathsf{pf\ self\ says}\ P}\ \textsc{Wf-tm-say}$$

$$\frac{S;E \vdash P : \mathsf{Prop}}{S;E \vdash \mathsf{pf}\ P : \mathsf{Type}}\ \textsc{Wf-tm-Pf} \qquad \frac{S;E \vdash p : P \quad S;E \vdash P : \mathsf{Prop}}{S;E \vdash \mathsf{return}_p\ p : \mathsf{pf}\ P}\ \textsc{Wf-tm-pf-ret}$$

$$\frac{S;E \vdash e_1 : \mathsf{pf}\ P \quad S;E \vdash e_2 : (x\!:\!P) \to \mathsf{pf}\ Q \quad x \notin \mathit{fv}(Q)}{S;E \vdash \mathsf{bind}_p\ e_1\ e_2 : \mathsf{pf}\ Q}\ \textsc{Wf-tm-pf-bind}$$

$$\frac{S;E \vdash v_1 : k \quad S;E \vdash v_2 : k \quad \mathit{atomic}\, S\, k \quad \mathit{val}(v_1) \quad \mathit{val}(v_2) \quad S;E, x \sim (v_1 = v_2)\!:\!k \vdash e_1 : t \quad S;E \vdash e_2 : t}{S;E \vdash \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}\ \textsc{Wf-tm-if}$$

$$\frac{S;E \vdash e : s \quad \mathit{converts}\ E\ s\ t}{S;E \vdash \langle e : t\rangle : t}\ \textsc{Wf-tm-cast}$$

Figure 1. AURA typing rules

```
data And :Prop → Prop → Prop {
|both :(p1:Prop) → (p2:Prop) → p1 → p2 → And p1 p2
}
```

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*. *False* itself is definable: it is a proposition with no constructors:

```
data False :Prop { }
```

Assertions, like the *MayPlay* proposition from Section 2, are uninhabited constants that construct Props:

```
assert MayPlay :prin → Song → Prop
```

While assertions are similar in flavor to datatypes with no constructors, there is a key difference. When an empty datatype is scrutinized by a match expression, the match may be assigned any type. Hence if we were to define *MayPlay* as an empty inductive type, *A* says *False* would follow from *A* says *MayPlay A freebird*. In contrast, there is no elimination form for assertions. This means that principals may sign assertions without compromising their says monad's consistency.

3.4 Core term typing

Type is the type of computation expressions, and Prop is the type of propositions. The constant Kind classifies both Type and Prop, as shown in rules WF-TM-TYPE and WF-TM-PROP. (Here and elsewhere, we use the lowercase word "type" to mean any classifier in the type system—Prop and Type are both "types" in this sense.)

The typechecking rules for constructors declared in the signature (WF-TM-CTR) and for free variables (WF-TM-FV) are completely standard. More interesting is WF-TM-ARR, which states that the type of an arrow is the type of arrow's output type. The latter is required to be one of Type, Prop, or Kind, which rules out nonsensical arrow forms. For example, $(x\!:\!\mathsf{Type}) \to \mathsf{Type}$ is legal whereas $(x\!:\!\mathsf{Type}) \to \mathsf{self}$ is not—the former could be the type of the polymorphic list constructor while the latter doesn't make sense since self is a computation-level value.

The WF-TM-ABS rule for introducing functions is standard except that, as in other lambda-cube like languages, AURA restricts what sorts of abstractions may be created. The argument to a function can be a term value, a proof, a type or a proposition. The resulting lambda must be typable with an arrow that itself has type Type or Prop. These restrictions imply that lambda abstractions may only be computational functions or proof terms. AURA does not support Type–level lambdas (as seen in F_ω) because doing so would require support for β-reduction at the type level. Such reductions, while useful for verification, appear superfluous here.

The interesting part of the WF-TM-APP rule is the side condition requiring either that t_2 is a value ($\mathit{val}(t_2)$) or that u does not depend on x ($x \notin \mathit{fv}(u)$). This restriction has the effect that, while AURA seems to be quite liberal with respect to the dependencies allowed by well-formed $(x\!:\!s) \to t$ terms, the actual dependencies admitted by the type system are quite simple. For instance, although the type system supports singleton types like S(0), it cannot check S(1+2) because the latter type depends on a non-value.

The upshot of these restrictions is that truly dependent types in AURA depend on values—i.e. terms that cannot reduce. While this limits the applicability of dependent types for program verification tasks, it greatly simplifies the metatheory, since there is no possibility of effectful computations appearing in a type.

Typechecking pattern match expressions is fairly standard (WF-TM-MATCHES), though it is a bit intricate due to AURA's support for a rich class of parameterized recursive datatypes. Only expressions that have saturated (fully applied) types can be matched against. The types of the branches must exhaustively match the constructors declared in the signature, and any parameters to the datatype being analyzed are also made available inside the branches. Each branch must return an expression of the same type, which is the result type of the entire match expression. Since datatypes and propositions in AURA may be nullary (have zero constructors), typechecking without inference requires that the match expression carry an annotation. The auxiliary definitions and the judgments used for typechecking the branches themselves can be found in Appendix B.

3.5 Principals and proofs

Principals are an integral part of access control logics, and AURA treats principals as first-class objects with type prin. The only built-in principal is self, which represents the identity of the currently running process (see WF-TM-PRIN and WF-TM-SELF); additional principal identifier constants could be accommodated by adding them with type prin, but we omit such a rule for simplicity's sake.

As described in Section 2, AURA uses the principal-indexed says monad to express access control policies. The proposition $a\,\mathsf{says}\,P$ means that principal a has asserted proposition P (either directly or indirectly). The expression $\mathsf{return}_s\ a\ p$ introduces proofs into the $a\,\mathsf{says}$ monad, and $\mathsf{bind}_s\ e_1\ e_2$ allows for reasoning under the monad. These constraints are shown in rules WF-TM-SAYS, WF-TM-SAYS-RET and WF-TM-SAYS-BIND. The rules are adapted from DCC [2], or more properly CDD [3], as AURA eschews DCC's label lattice in favor of explicit delegation among principals.

The expression $\mathsf{sign}(a, P)$ witnesses the assertion of proposition P by principal a (WF-TM-SIGN). Since $\mathsf{sign}(a, P)$ is intended to model evidence manufactured by a without justification, it should never appear in a source program. Moreover, since signed propositions are intended to be distributed and thus may escape the scope of the running AURA program, they are required to be closed. Note, however, that the declaration signature *S* must be available in whatever context the signature is to be ascribed meaning. In practice, this means that two distributed AURA programs that wish to exchange proofs need to agree on the signatures used to construct those proofs.

$$\frac{}{\textit{converts}\ E\ t\ t}\ \textsc{Conv-refl} \qquad \frac{\textit{converts}\ E\ t\ s}{\textit{converts}\ E\ s\ t}\ \textsc{Conv-symm}$$

$$\frac{\textit{converts}\ E\ s\ u \quad \textit{converts}\ E\ u\ t}{\textit{converts}\ E\ s\ t}\ \textsc{Conv-trans}$$

$$\frac{x \sim (s = t){:}k \in E}{\textit{converts}\ E\ s\ t}\ \textsc{Conv-axiom}$$

$$\frac{\textit{converts}\ E\ s_1\ t_1 \quad \textit{converts}\ E\ s_2\ t_2}{\textit{converts}\ E\ (s_1\ s_2)\ (t_1\ t_2)}\ \textsc{Conv-app}$$

$$\frac{\textit{converts}\ E\ s_1\ t_1 \quad \textit{converts}\ E\ s_2\ t_2}{\textit{converts}\ E\ (\lambda x\!:\!s_1.\ s_2)\ (\lambda x\!:\!t_1.\ t_2)}\ \textsc{Conv-abs}$$

$$\frac{\textit{converts}\ E\ s_1\ t_1 \quad \textit{converts}\ E\ s_2\ t_2}{\textit{converts}\ E\ ((x\!:\!s_1) \to s_2)\ ((x\!:\!t_1) \to t_2)}\ \textsc{Conv-arr}$$

Figure 2. Conversion

Creating $\mathsf{sign}(a, P)$ requires a's authority. AURA models the authority of a running program with the principal constant self. The $\mathsf{say}\,P$ operation creates an object of type $\mathsf{pf}\ \mathsf{self}\,\mathsf{says}\,P$. Intuitively, this operation creates the signed assertion $\mathsf{sign}(\mathsf{self}, P)$ and injects it as a proof term for further manipulation (see WF-TM-SAY).

AURA uses the constant $\mathsf{pf} : \mathsf{Prop} \to \mathsf{Type}$ to wrap the access control proofs that witness propositions as program values, as shown in the rule WF-TM-PF. The pf type operates monadically: $\mathsf{return}_p\ p$ injects a proof p into the term level and bind_p allows a computation to compose proofs (rules WF-TM-PF-RET and WF-TM-PF-BIND). Such a separation between proofs and computations is necessary to prevent effectful program expressions from appearing in a proof term. For example, if $\mathsf{say}\,P$ was given type $\mathsf{self}\,\mathsf{says}\,P$ rather than $\mathsf{pf}\ (\mathsf{self}\,\mathsf{says}\,P)$, it would be possible to create a bogus "proof" $\lambda x\!:\!\mathsf{Prop}.\ \mathsf{say}\,x$; the meaning of this "proof" would depend on the authority (self) of the program that applied the proof object.

3.6 Equality and conversion

Some typing rules (e.g. WF-TM-APP) require checking that two terms can be given the same type. Satisfying such constraints in a dependently typed language requires deciding when two terms are equal—a difficult static analysis problem in the best case.

In AURA we address this with a conditional construct. Dynamically, $\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ steps to e_1 when v_1 and v_2 are equal, otherwise the expression steps to e_2. Statically (rule WF-TM-IF), the then branch is typed in an environment containing the static constraint ($v_1 = v_2$). As we will see shortly, the constraint may be used to perform safe typecasts. This is an instance of the type refinement problem, well known from pattern matching in languages such as Coq [17], Agda [32], and Epigram [29].

AURA limits its built-in equality tests to inhabitants of *atomic* types. The built-in prin type is atomic, as is any type defined by a non-parameterized Type declaration, each of whose constructors takes no arguments. The *List* type is not atomic, nor is *List nat* (since *cons* takes an argument). However, the following *Song* type is atomic:

$$\mathsf{data}\ \textit{Song}\!:\ \mathsf{Type}\ \{\ |\textit{freebird}\!:\ \textit{Song}\ \ |\textit{ironman}\!:\ \textit{Song}\ \}$$

In other words, atomic types are prin and enumerated types. Our definition of atomic type is limiting, but we believe it can be naturally extended to first-order datatypes.

With equalities over atomic types in the context, we can now consider the issue of general type equality. As in standard presentations of the Calculus of Constructions [10] we address type equality in two acts.

Two types in Aura are considered equivalent when they are related by the conversion relation. This relation, written *converts* and defined in Figure 2, is of course reflexive, symmetric, and transitive; the key rule is Conv-axiom, which uses equality assumptions in the environment. For instance, under assumption $x = \mathsf{self}$, term $x\ \mathsf{says}\ P$ converts to $\mathsf{self}\ \mathsf{says}\ P$. As equalities only mention atomic values, conversion will only alter the "value" parts of a type—convertible types always have the same shape up to embeded data values.

Aura contains explicit, safe typecasts. As specified in rule Wf-tm-cast, term $\langle e : T\rangle$ is assigned type T whenever e's type is convertible with T. Many standard presentations of dependently typed languages use implicit conversions, which may occur anywhere in a type derivation, but the explicit cast is appealing as it gives an algorithmic type system. Casts have no run-time effect and are simply discarded by our operational semantics.

3.7 Evaluation rules

Figure 3 defines Aura's operational semantics using a call-by-value small-step evaluation relation.

Most of the evaluation rules are straightforward. The rule Pf-bind is a standard beta reduction for monads. The term $\mathsf{say}\,P$ creates a proof that principal self has asserted that proposition P is true; therefore, it evaluates to an assertion "signed" by principal self. There are two possibilities in the evaluation of $\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$: when v_1 equals v_2, it evaluates to e_1, otherwise it evaluates to e_2. We define two auxiliary reduction relations to implement the reduction rule for pattern matching.

We write $(v, b) \mapsto_b e$ to denote the evaluation of a value v against a set of branches. These evaluation rules search through the list of branches until v matches with the constructor of one of the branches, at which point the rules focus on the branch and supply the body of the branch with the arguments in v. The tricky part lies in correctly identifying the arguments in v and discarding the type parameters. We write $(v, c, \mathit{body}) \mapsto_c (e, n)$ to denote the evaluation of the body of the branch where v matches with the constructor c in the branch. Here, n is the number of parameters that should be discarded before the first argument of v is found. For example, the first parameter *nat* in the value *cons nat 3* (*nil nat*) of type *List nat* has no computational content; therefore it should be discarded during the evaluation of pattern matching. Note that the semantics represents constructors as a pair of the constructor name c and its number of type parameters. For instance, in the definition of polymorphic lists shown previously, the representation of *cons* is (*cons, 1*).

$$\boxed{t \mapsto t'}$$

$$\frac{\mathit{val}(v)}{(\lambda x{:}t.\ e)\ v \mapsto \{v/x\}e}\ \textsc{App}$$

$$\frac{}{\mathsf{bind}_p\ (\mathsf{return}_p\ e_1)\ e_2 \mapsto e_2\ e_1}\ \textsc{Pf-bind}$$

$$\frac{}{\mathsf{say}\ P \mapsto \mathsf{return}_p\ (\mathsf{sign}(\mathsf{self}, P))}\ \textsc{Say}$$

$$\frac{v_1 = v_2}{\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_1}\ \textsc{If-eq}$$

$$\frac{v_1 \neq v_2}{\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_2}\ \textsc{If-neq}$$

$$\frac{\mathit{val}(v)}{\langle v : t\rangle \mapsto v}\ \textsc{Cast}$$

$$\frac{(v, \mathit{branches}) \mapsto_b e}{\mathsf{match}\ v\ t\ \mathsf{with}\ \{\mathit{branches}\} \mapsto e}\ \textsc{Match}$$

$$\boxed{(v, b) \mapsto_b e}$$

$$\frac{(v, c, \mathit{body}) \mapsto_c (e, 0)}{(v, \mathit{brn}\ c\ \mathit{body}\ \{\mathit{rest}\}) \mapsto_b e}\ \textsc{B-here}$$

$$\frac{(v, \mathit{rest}) \mapsto_b e}{(v, \mathit{brn}\ c\ \mathit{body}\ \{\mathit{rest}\}) \mapsto_b e}\ \textsc{B-earlier}$$

$$\boxed{(v, c, \mathit{body}) \mapsto_c (e, n)}$$

$$\frac{}{((c, n), (c, n), \mathit{body}) \mapsto_c (\mathit{body}, n)}\ \textsc{Ctr-base}$$

$$\frac{\begin{array}{l}\mathit{val}(v_2) \quad m > 0\\ (v_1, (c, n), \mathit{body}) \mapsto_c (\mathit{body}, m)\end{array}}{(v_1\ v_2, (c, n), \mathit{body}) \mapsto_c (\mathit{body}, m-1)}\ \textsc{Ctr-param}$$

$$\frac{(v_1, (c, n), \mathit{body}) \mapsto_c (e, 0)}{(v_1\ v_2, (c, n), \mathit{body}) \mapsto_c (e\, v_2, 0)}\ \textsc{Ctr-arg}$$

Figure 3. Reduction Rules

3.8 Metatheory

We have proved soundness (in terms of progress and preservation) for Aura. The proofs are fully mechanized in the Coq proof assistant.

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* $\mathit{val}(e)$ *or exists* e' *such that* $e \mapsto e'$.

We have also proved that typechecking in Aura is decidable.

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 have mechanized all parts of these decidability results by giving constructive proofs of the form $\phi \vee \neg\phi$. For instance, the constructive proof of $(S \vdash \diamond) \vee \neg(S \vdash \diamond)$ is a total algorithm that decides signature well-formedness.

For ease of explanation, the judgments and rules presented in this section are a close approximation of the formal definitions of Aura. For instance, in order to prove the preservation of pattern matching, we have to take the parameters and arguments supplied to the constructor in the pattern matching evaluation rules. In order to prove the decidability of typechecking, we strengthened the typing judgments to take two signature arguments: one contains the type declarations of the top-level type constructors (e.g., *List*) that can appear in mutually recursively defined datatypes and the other is used for looking up the constructors (e.g., *nil*, *cons*) of the top-level type constructors. However, this simplified presentation has the same key invariants as the full type system. The full detailed version of the type system can be found in the Appendices.

4. Validation and Prototype Implementation

Mechanized proofs AURA has 20 reduction rules, 40 typing judgments (including the well-formedness of terms, environments and signatures), and numerous other relations such as atomic equality types to constrain the type system. For a system of this size, implementing a fully complete, mechanized version of the soundness proofs is challenging.

We formalized the proofs of soundness and the decidability of typechecking for AURA in the Coq proof assistant[3]. We use a variant of the locally nameless representation [9] to formalize the metatheory of the language. Well documented definitions of AURA including typing rules, reduction rules, and other related relations require about 1400 lines of Coq code. The soundness proofs take about 6000 lines of Coq code, and the proofs of the decidability of typechecking take about 5000 lines of Coq code. The automation used in the these Coq proofs is relatively rudimentary; we did not devote much time to writing automation tactics.

The most intricate parts of the language design are the invariants of the inductive datatypes, the dependent types, atomic equality types, and the conversion relations. This complexity is reflected in the Coq proof development in two ways: one is in the number of lemmas stating the invariants of the datatype signatures, the other is in the number of revisions made to the Coq proofs due to design changes motivated by failure to prove soundness. We found that for such a complicated system, mechanized proofs are well-suited for dealing with design iteration, as Coq can easily identify which proofs require modification when the language design changes.

Because AURA is a superset of system F with inductively defined datatypes, we conjecture that, without much difficulty, we could extract mechanized soundness proofs of other related type systems from the Coq proofs of AURA.

Typechecker and interpreter The prototype AURA typechecker and interpreter together implement the language as it is formalized in Coq with only minor differences. The typechecker recognizes a small number of additional types and constants that are not present in the formal definition, including literal 32-bit integers, literal strings and tuples. Although it is derivable in AURA, we include a fix constant for defining recursive functions; by using this constant together with tuples, mutually recursive functions can be defined more succinctly than is possible in the formal definition. To allow for code reuse, we have added an include statement that performs textual substitution from external files. The software sorts included files in dependency order and copies each only once.

AURA is not meant for general-purpose application development; instead, it is designed to be used synergistically with existing production programming languages. One way we plan to take to reach this goal is to eventually target the .NET runtime, as it encourages language intermingling (see Section 7). We plan to expose authorization polices written in AURA to the .NET common type system by providing libraries for interacting at runtime with propositions. We will also explore the possibilities of rewriting annotated methods in compiled .NET code to make implicit calls to these libraries. This approach allows any language that uses the common type system to interoperate with AURA.

5. An Extended Example

In this section, we illustrates the key features of AURA's type system by explaining a program implementing a simple streaming music server.

The extended code sample is given in Figures 4 and 5. The example program typechecks in the prototype AURA interpreter and uses some of the language extensions discussed in Section 4. On line 1) the program imports library code that defines utility types (such as dependent tuples and lists).

We imagine that the server implements a policy in which every song may have one or more owners, corresponding to principals who intrinsically have the right to play the song. Additionally, song owners may delegate their listening rights to other principals.

This policy is defined over predicates *Owns* and *MayPlay*, which are declared as assertions in lines 5 and 6. Recall that assertions are appropriate because we cannot expect to find closed proofs of ownership and delegation in pure type theory.

The main policy rule, *shareRule* (line 12) is defined using a say expression. The type of *shareRule* is an implication wrapped in two monads. The outer pf monad is required because say accesses a private key and must be treated effectfully. The inner self says monad is required to track the provenance of the policy. The implication encodes the delegation policy above. This rule provides a way to build up a value of type pf (self says (*MayPlay a s*)), which is required before *a* can play song *s*.

The exact form of *shareRule* is somewhat inconvenient. We derive two more convenient rules, $shareRule'$ and $shareRule''$ (lines 53 and 76). These rules use monadic bind and return operations to change the placement of pf and says type constructors relative to *shareRule*'s type. The resulting type of $shareRule''$ shows that one can obtain a proof of pf (self says (*MayPlay a s*)) by a simple application of $shareRule''$ to various arguments, as shown in line 101.

The key functionality of the music server is provided by a function stub, *playFor*, which is intended to model an effectful function that streams a provided song to a specified principal. Its type is given by the annotation on line 20. The *playFor* function takes the song to be played and the principal it should play on behalf of as its first two arguments. The third argument is a proof of the proposition self says (*MayPlay A s*), demonstrating the requesting principal's capability to play the song, which is required by the server's policy. As modeling an audio API would clutter the example, *playFor* simply returns a unit value. In a real implementation, *playFor* would call into the trusted computing base, which would also log appropriate proofs for future auditing.

The remaining code implements the application's main functionality. The *handleRequest* function takes a delegation request and, using a provided database of owner information, attempts to construct an appropriate self says *MayPlay* proof. If it succeeds, *playFor* is invoked.

The implementation of *handleRequest* (line 93) is straightforward. There are two interesting things to note. First, *handleRequest* takes a database of owner information expressed as a list of *OwnerRecord*s. *OwnerRecord* (line 8) is an inductive type whose single constructor has a dependent type. Because *ownerRecord*'s third argument depends on its first two, *OwnerRecord* encodes an existential type. Second, the match expression on line 98 relies on the fact that (*getOwnerProof s o l*) returns an object of type *Maybe* (pf (self says (*Owns p s*))). Getting such a type is possible because, when *getOwnerProof* pulls a proof from the list, its type is refined so that the existentially bound principal and song are identified with *p* and *s*.

GetOwnerProof (line 30) performs this type refinement in several steps. It uses the fixpoint combinator (line 33) to perform a list search. After each *OwnerRecord* is decomposed, we must check its constituent parts to determine if it is the correct record and, if so, refine the types appropriately. The action occurs between lines 42 and 48. At runtime the first if expression tests for dynamic equality between the principal we're searching for, *p*, and the principal store in the current record, p'. A similar check is performed for between *Song*s *s* and s'. If both checks succeed then we cast *proof*:pf (self says *Owns* $p' s'$) to type pf (self says *Owns p s*) and

[3] Code available at: http://www.cis.upenn.edu/~stevez/sol/

```
include ”tuple.core” include ”list.core”

data Song :Type { |freebird: Song |ironman: Song }

assert Owns :prin → Song → Prop;
assert MayPlay :prin → Song → Prop;

data OwnerRecord :Type {
|ownerRecord :(p: prin) → (s: Song) →
                         (pf (self says (Owns p s))) → OwnerRecord }

let shareRule :
    pf (self says ((o: prin) → (r: prin) → (s: Song) →
        (Owns o s) → (o says (MayPlay r s)) → (MayPlay r s))) =
  say ((o: prin) → (r: prin) → (s: Song) →
        (Owns o s) → (o says (MayPlay r s)) → (MayPlay r s))
in

(∗ A real implementation would do something here ∗)
let playFor :(s: Song) → (p: prin) →
                               (pf (self says (MayPlay p s))) → Unit =
  λs: Song . λp: prin . λproof: (pf (self says (MayPlay p s))) . unit
in

let notFound :(p: prin) → (s: Song) →
                               (Maybe (pf (self says (Owns p s)))) =
  λp: prin. λs: Song. nothing (pf (self says Owns p s))
in

let getOwnerProof: (s: Song) → (p: prin) →
        (List OwnerRecord) → (Maybe (pf (self says (Owns p s)))) =
  λs: Song . λp: prin . λownerRecords: List OwnerRecord .
    fix (λrec: (List OwnerRecord) →
                 (Maybe (pf (self says (Owns p s)))).
        λl: (List OwnerRecord) .
          match l with (Maybe (pf (self says Owns p s))) {
          |nil → notFound p s
          |cons → λx:OwnerRecord. λxs: List OwnerRecord .
               match x with (Maybe (pf (self says Owns p s))) {
               |ownerRecord → λp′:prin. λs′:Song.
                                 λproof: pf (self says (Owns p′ s′)).
                 if p = p′
                   then if s = s′
                     then
                       just (pf (self says (Owns p s)))
                       ⟨proof: (pf (self says (Owns p s)))⟩
                     else rec xs
                   else rec xs
               } } )
        ownerRecords
in

let shareRule′ :
    (pf ((o: prin) → (r: prin) → (s: Song) →
        (self says (Owns o s)) → (o says (MayPlay r s)) →
        (self says (MayPlay r s)))) =
  bind shareRule (λsr: (self says
                          ((o: prin) → (r: prin) →
                          (s: Song) → (Owns o s) →
                          (o says (MayPlay r s)) →
                          (MayPlay r s))) .
    return (λo: prin. λr: prin. λs: Song.
             λowns: (self says (Owns o s)).
             λmay: (o says (MayPlay r s)).
      bind sr (λsr′: ((o′: prin) → (r′: prin) → (s′: Song) →
                        (Owns o′ s′) → (o′ says (MayPlay r′ s′)) →
                          (MayPlay r′ s′)) .
      bind owns (λowns′ :(Owns o s).
                     return self (sr′ o r s owns′ may)))))
in
```

Figure 4. AURA code for a music store (cont. in Figure 5).

```
let shareRule′′: (o: prin) → (p: prin) → (s: Song) →
            (pf self says (Owns o s)) →
            (pf (o says (MayPlay p s))) →
            (pf self says (MayPlay p s)) =
  λo: prin. λp: prin. λs: Song.
    λownsPf: pf (self says (Owns o s)).
    λplayPf: pf (o says (MayPlay p s)).
      bind ownsPf (λopf: (self says (Owns o s)).
      bind playPf (λppf: (o says (MayPlay p s)).
      bind shareRule′ (λsr′:
            ((o′: prin) → (r′: prin) → (s′: Song) →
              (self says (Owns o′ s′)) →
              (o′ says (MayPlay r′ s′)) →
              (self says (MayPlay r′ s′))) .
          (return (sr′ o p s opf ppf)))))
in

let handleRequest: (s: Song) → (p: prin) → (o: prin) →
                         (List OwnerRecord) →
                         (delPf: pf (o says (MayPlay p s))) → Unit =
  λs: Song. λp: prin. λo: prin. λl: List OwnerRecord.
    λdelPf: pf (o says (MayPlay p s)).
    match (getOwnerProof s o l) with Unit {
    |nothing → unit
    |just → λx: (pf (self says (Owns o s))).
                       playFor s p (shareRule′′ o p s x delPf)
    }
in unit
```

Figure 5. AURA code for a music store (cont. from Figure 4).

return it packaged as a *Maybe*. If either dynamic check fails we repeat again, and, if no match is found, we eventually return *Nothing*.

6. Related Work

We have published related results on AURA$_0$, a language closely related to the Prop fragment of AURA [39]. This includes soundness and strong normalization proofs for AURA$_0$, as well as discussion and examples of audit in the presence of authorization proofs.

One intended semantics for AURA implements objects of form sign(A, P) as digital signatures. All cryptography occurs at a lower level of abstraction than the language definition. This approach has previously be used to implement declarative information flow policies [40]. An alternative approach is to treat keys as types or first class objects and to provide encryption or signing primitives in the language [7, 15, 35, 27, 26]. Such approaches typically provide the programmer with additional flexibility but complicate the programming model.

Authorization logics Many logics and languages [5, 6, 14, 11, 21, 2, 20] have tracked authorization using says. We follow the approach of DCC [2], a logic in which says is defined as an indexed monad. This is compelling for several reasons. First, DCC proofs are lambda-terms, a fact we exploit to closely couple the Prop and Type universes. Second, DCC is a strong logic and important authorization concepts, such as the *acts-for* relation and the hand-off rule $(A$ says B *acts-for* $A) \rightarrow (B$ *acts-for* $A)$, can be defined or derived. Third, DCC is known to enjoy a non-interference property: in the absence of delegation, statements in the A says monad will not affect the B says monad. In our setting this means that a given program cannot be tricked by what an untrusted program says. AURA modifies DCC in several ways. In addition to adding dependent types, AURA omits DCC's protects relation. The protects relation strengthens monadic bind, making propositions A says $(B$ says $P)$ and B says $(A$ says $P)$ interderivable. While useful in other settings, such equivalences appear incorrect for access control. Additionally AURA's use of signatures changes some meta-theoretic properties of DCC leading to, for example, a more subtle proof of normalization [39].

The Grey project [11] uses proof carrying authorization in a manner similar to AURA. In Grey, mobile phone handsets build authorization proofs that unlock doors. While AURA is a unified authorization logic and computation language, Grey's logic is not integrated with a computation language.

DeYoung, Garg, and Pfenning [20] describe a constructive authorization logic that is parameterized by a notation of time. Propositions and proofs are annotated with time intervals during which they may be judged valid. This allows revocation to be modeled as credential expiration.

Language-based access control The trust management system PolicyMaker [13] treats the handling of access control decisions as a distributed programming problem. A PolicyMaker *assertion* is a pair containing a function and (roughly) a principal. In general, assertion functions may communicate with each other, and each function's output is tagged by the associated principal. PolicyMaker checks if a request complies with policy by running all assertion functions and seeing if they produce an output in which a distinguished principal POLICY says "approve". Principal tags appear similar is purpose, but not realization, to says in AURA. Note also that expressing security properties via term-level computation is fundamentally different from expressing them as types, the approach followed in other work discussed here. The ideas in PolicyMaker have been refined in KeyNote [13] and REFEREE [16].

Fournet, Gordon and Maffeis [21, 22] discuss authorization logic in the context of distributed systems. They use a limited form of dependent pairs to associate propositions with data. Unlike in AURA, proofs are erased at runtime. Consequently, their type discipline is best suited for closed systems that do not require high-assurance logging.

The Fable language [37] associates security labels with data values. Labels may be used to encode information flow, access control, and other policies. Technically, labels are terms that may be referred to at the type level; *colored* judgments separate the data and label worlds. The key security property is that standard computations (i.e. application computations described with color *app*) are parametric in their labeled inputs. Unlike AURA proofs, the label sub-language (i.e. policy computations described with color *pol*) admits arbitrary recursion. Hence the color separation may restrict security sensitive operations to a small trusted computing base, but does not give rise to a logical soundness property.

Dependent type theory The AURA language design was influenced by dependent type systems like the Calculus of Constructions (CoC) [10, 18]. Both CoC and AURA contain dependent types and a unified syntax encompassing both types and terms. However there are several important differences between CoC and AURA. Most critically, CoC's type equality includes beta equivalence but AURA's does not. Type-level beta reduction, while convenient for verification, is unnecessary for expressing authorization predicates, and greatly complicates language design and use.

As realized in the Coq proof assistant [17], CoC can contain inductive types and different universes for computation and logic types—AURA universes Prop and Type correspond to Prop and Set in Coq. However, because Set is limited to pure computations, Coq does not need AURA's pf mechanism to separate Prop from Set. In Coq all inductive declarations are subject to a complex *positivity* constraint which ensures inductive types have a well-defined logical interpretation. By contrast, AURA uses a simpler positivity constraint in Prop and no constraint in Type. Additionally, AURA permits less type refinement than Coq does for type indices—Coq datatypes can be used to encode GADTs. Compared with Coq, AURA is strictly weaker for defining logical predicates, but is more expressive for defining datatypes for use in computation.

Several other projects have combined dependent types and pragmatic language design. Ynot (an embedding of Hoare Type Theory [31] in Coq), Agda [32], and Epigram [29] are intended to support general purpose program verification and usually require that the programmer construct proofs interactively. By contrast, Dependent ML [46], ATS [46, 45], and RSP1 [43] provide distinguished dependency domains and can only express constraints on objects from these domains. These dependency domains are intended to be amenable to automated analysis. Cayenne [8] extends Haskell with general purpose dependent types. In Cayenne, type equality is checked by normalizing potentially divergent Haskell terms, a strategy which may cause typechecking itself to diverge. Hancock and Setzer [23] present a core calculus for interactive programming in dependent type theory; their language uses an IO monad to encapsulate stateful computations. Inhabitants of the monad are modeled as imperative programs and type equality is judged up to a bisimulation on (imperative) program text.

Peyton Jones and Meijer describe the Henk typed intermediate language [25]. Henk is an extension of the lambda cube family of programming languages that includes CoC. Like AURA, Henk is intended to be a richly-typed compiler intermediate language. Unlike AURA, Henk has not been proved sound. Additionally, its lack of a pf monad (or equivalent technique for isolating computations from proofs) makes it unsuitable for programming in the presence of both dependent types and effects.

7. Future Work

Future work: Theory One important direction for future work is to study AURA's security properties, such as the non-interference of AURA's authorization logic and the strong normalization of AURA's authorization proofs. The non-interference property of authorization logics (c.f. [2]) allows analysis on the influence of one principal's assertions to other principals' beliefs; this provides more confidence in the correctness of authorization logics. Since AURA used DCC as the authorization logic, we believe a proof similar to that presented in [2] is also possible for AURA.

AURA allows programmers to specify secure functions such as *playFor* to protect access to resources. As a corollary of the type safety theorem, these functions cannot be invoked unless a valid proof is provided. To ensure that proofs are meaningful and can later be compared during auditing, we would like to prove that every reduction of a proof will result in the same normal form. In our previous work [39], we have proved that a simplified Prop fragment of AURA is strongly normalizing and confluent. As future work, we intend to prove the same results for the full Prop fragment of AURA using similar techniques.

As discussed in Section 3, AURA's restrictions on dependency and its weak notion of type refinement allow for decidable typechecking. We believe that there may be other decidable, but more-liberal, variants of the type system. As we continue to write AURA programs, we expect to discover precisely what relaxations of the type system would be useful to programmers. Additionally, the literature on GADTs [33] and dependent pattern matching [32] promises to be a fertile source of inspiration and examples. Future versions of AURA may be extended along these lines.

Likewise, while making the pf construct a monad was a convenient and effective design decision, we might be able to relax its treatment. For instance, pf's elimination form is currently monadic bind, but it is probably safe to instead eliminate pf by pattern matching when constructing arbitrary objects in Type. We conjecture that this does not affect the soundness of AURA, and we plan to explore such alternative designs in the future.

Section 2 describes the correspondence between a says P and objects digitally signed by a's private key. It is natural, then, to wonder about the possibility of an analog for public key encryption—perhaps terms of type T for a could be constructed from objects of type T encrypted with a's public key. It is unclear how to integrate such an additional construct with AURA, not least because while the says monad makes complete sense operating only at the Prop level, we almost certainly want to encrypt data of kind Type. Additionally, our use of dependent types means that the type of an AURA term will often reference part of the term, which may well be unacceptable for data that is meant to be encrypted.

The tracking of information flow was one of the first uses proposed for DCC [4], and even without encryption AURA's assertions are sometimes reminiscent of confidentiality tracking; were encryption to be added, the similarities would be even more pronounced. It may be possible to take advantage of this by equipping AURA with a more general notion of information flow—which does not necessarily have as straightforward a cryptographic interpretation—for use internal to a single well-typed application while reverting to the coarser-grained says (and possibly for) when communication with the outside world is desired. The challenge, of course, is to make this change of granularities as fluid and light as possible.

Even without information flow it may still be useful to have a better idea of which proofs may come from the outside world. After all, operations on digital signatures are not trivial, but since proofs are defined to be computation-free, a purely local proof could be given a more efficient but less portable representation, and certain proofs might be completely elided at runtime. For the first case, we would first need to extend our formalism with some notion of network communication; inference could be performed backwards from communication points to ascertain which proofs need not be represented in portable form. As an initial step towards recognizing the second case, we might consider an additional form of abstraction, with an argument that cannot be used in certain ways but is guaranteed to be necessary only at compile time; ideally, however, we would want to infer these abstractions during compilation.

It is clear from our examples that AURA is fairly verbose. As it is meant to be an intermediate language, this is not a pressing usability issue. We hope that a higher-level language that generates AURA will be able to cut down on this verbosity using inference techniques. Our proof-passing style also suggests the use of some variety of proof inference. Of course, this very quickly becomes undecidable, but that does not rule out practical partial solutions.

Finally, although AURA emphasizes the security aspects of programming with an embedded authorization logic, there might be other applications of this idea. In particular, one of the challenges of making program verification via dependent types practical is the need to construct and otherwise manipulate proof objects. Of course, one can always add axioms to the logic, but doing so can easily compromise its consistency. Failures due to a poor choice of axioms might be hard to isolate when debugging. The says monads of DCC provide a possible intermediate ground: One could imagine associating a principal with each module of the program and then allowing modules to make assertions. Explicit trust delegations would then be required when importing axioms from one module to another; such delegations would document the module dependencies and help the typechecker isolate uses of faulty axioms. We speculate that it is even possible that blame (in the style similar to that proposed by Wadler and Findler [42]) can be appropriately assigned to offending modules whenever a run-time error caused by incorrect assertions is encountered.

Future work: Practice The single-step interpreter is useful as a tool for checking the correctness of small examples; however, it is infeasible to use it to run code in a production environment. As such, we are extending the implementation to generate CIL bytecode compatible with both Microsoft's .NET CLR and the open-source Mono runtime. Don Syme's work on ILX aids us greatly in this effort. ILX, described in [38], is a group of extensions to the CIL that facilitates the use of higher-order functions, discriminated unions and parametric polymorphism. By compiling for this existing standard execution environment, we will gain access to the ecosystem of .NET software and libraries. Most notably, we should be able to make use of existing code for cryptography and cross-platform networking. We will also be free from having to worry about lower-level issues like efficient machine code generation and garbage collection, both of which are well outside of the AURA project's scope.

Additionally, there remain practical issues that AURA must address in order to fully express policies likely to be found in its intended problem domain. Chief among these is the demand for the signatures that *expire*, either due to explicit revocation or simply the passage of time. This stands in contrast to our current formalism—and, indeed, most formalisms of programming languages, as a term that successfully typechecks is generally seen as valid for regardless of the time or the state of the world. It would, of course, be possible to define the operational semantics of AURA such that every operation has a chance to fail at runtime due to digital signature expiration, but this could easily make programming quite cumbersome. Instead, we hope to find a solution that allows time and revocation to be referenced by AURA in an intuitive way; one possibility, explored by Garg and Pfenning [20], is the use of *linear* logic, which is naturally suited to describing resources that can, in some sense, be used up.

References

[1] Martín Abadi. Logic in access control. In *Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03)*, pages 228–233, June 2003.

[2] Martín Abadi. Access control in a core calculus of dependency. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 263–273. ACM, 2006.

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

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

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

[7] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically masked information flows. In *Proceedings of the International Static Analysis Symposium*, LNCS, Seoul, Korea, August 2006.

[8] Lennart Augustsson. Cayenne–a language with dependent types. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 239–250, September 1998.

[9] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2008.

[10] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors,

Handbook of Logic in Computer Science, volume 2, pages 117–309. Clarendon Press, Oxford, 1992.

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

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

[13] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999.

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

[15] Tom Chothia, Dominic Duggan, and Jan Vitek. Type based distributed access control. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, Asilomar, Ca., USA, July 2003.

[16] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust management for web applications. *Computer Networks and ISDN Systems*, 29:953–964, 1997.

[17] The Coq Development Team, LogiCal Project. *The Coq Proof Assistant Reference Manual*, 2006.

[18] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76, 1988.

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

[20] Henry DeYoung, Deepak Garg, and Frank Pfenning. An authorization logic with explicit time. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF-21)*, Pittsburgh, June 2008.

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

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

[23] Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, pages 317–331, London, UK, 2000. Springer-Verlag.

[24] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindly, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980.

[25] Simon Peyton Jones and Erik Meijer. Henk: A typed intermediate language. In *Proceedings of the Types in Compilation Workshop*, Amsterdam, June 1997.

[26] Peeter Laud. On the computational soundness of cryptographically masked flows. *SIGPLAN Not.*, 43(1):337–348, 2008.

[27] Peeter Laud and Varmo Vene. A type system for computationally secure information flow. In *Proceedings of the 15th International Symposium on Fundamentals of Computational Theory*, volume 3623, pages 365–377, Lübeck, Germany, 2005.

[28] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–184, 2007.

[29] Conor McBride. *The Epigram Prototype: a nod and two winks*, April 2005. Available from `http://www.e-pig.org/downloads/epigram-system.pdf`.

[30] Andrew C. Myers, Stephen Chong, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. 1999.

[31] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *Proc. 11th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2006.

[32] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[33] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 2006.

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

[35] Geoffrey Smith and Rafael Alpízar. Secure information flow with random assignment and encryption. In *Proceedings of The 4th ACM Workshop on Formal Methods in Security Engineering: From Specifications to Code (FSME'06)*, pages 33–43, November 2006.

[36] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

[37] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2008.

[38] Don Syme. ILX: Extending the .NET Common IL for functional language interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[39] Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Steve Zdancewic. Evidence-based audit. In *Proc. of the IEEE Computer Security Foundations Symposium*, 2008. Extended version available as U. Pennsylvania Technical Report MS-CIS-08-09.

[40] Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, pages 192–206, Berkeley, California, 2007.

[41] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995. Some errata fixed August 2001.

[42] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, pages 15–26, 2007.

[43] E. Westbrook, A. Stump, and I. Wehrman. A language-based approach to functionally correct imperative programming. In B. Pierce, editor, *10th ACM SIGPLAN International Conference on Functional Programming*, Tallinn, Estonia, 2005.

[44] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Trans. Comput. Syst.*, 12(1):3–32, 1994.

[45] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[46] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, San Antonio, Texas, September 1998.

A. AURA's Signatures

A.1 Syntax

The formal definitions for AURA's signatures are as follows.

Constructor Decls	*cdecls*	$::=$	$\cdot \mid cdecls \,\vert\, ctr : t$
Data Decl	*ddecl*	$::=$	$\mathsf{data}\ ctr : t\ \{cdecls\}$
Definitions	*defns*	$::=$	$ddecl \mid defns\ \mathsf{with}\ ddecl$
Assertions	*assn*	$::=$	$\mathsf{assert}\ ctr : t$
Bundle	*bundle*	$::=$	$defns \mid assn$
Signature	*S*	$::=$	$\cdot \mid S, bundle$

An AURA signature *S* is a list of bundles. Each bundle consists of either an assertion or a list of mutually recursively defined datatype declarations. Each datatype declaration is a tuple of the type constructor name, its type, and a list of its data constructor declarations. Each data constructor declaration is itself a pair of the data constructor's name and its type.

A.2 Typing rules for signatures

In Figure 6, we present the auxiliary definitions used by the main typing judgments for well-formed signatures.

$$\boxed{positive\ ctrs\ t}$$

$$\frac{}{positive\ ctrs\ ctr} \qquad \frac{positive\ ctrs\ t_1 \quad ctrs \cap ctrs\_of\ t_2 = \emptyset}{positive\ ctrs\ t_1\ t_2}$$

$$\frac{positive\ ctrs\ t_2 \quad ctrs \cap ctrs\_of\ t_1 = \emptyset}{positive\ ctrs\ (x : t_1) \to t_2}$$

$$\boxed{wf\_dom(defns)}$$

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

$$\frac{\begin{array}{c} wf\_dom(defns) \quad dom(cdecls) \cap dom(defns) = \emptyset \\ ctr \notin dom(defns) \quad ctr \notin dom(cdecls) \end{array}}{wf\_dom((defns\ \mathsf{with\ data}\ ctr : t\ \{cdecls\}))}$$

$$\boxed{get\_tctr\_defns(defns)}$$

$$\frac{}{get\_tctr\_defns(\cdot) = \cdot}$$

$$\frac{get\_tctr\_defns(defns) = defns'}{\begin{array}{l} get\_tctr\_defns(defns\ \mathsf{with\ data}\ c : t\ \{cdecls\}) \\ = defns'\ \mathsf{with\ data}\ c : t\ \{\cdot\} \end{array}}$$

Figure 6. Auxiliary Definitions

To ensure the consistency of the Prop fragment, the data types in the Prop universe are subject to positivity check. We write *positive ctrs t* to denote that the set of type constructors *ctrs* only appear positively in type t. AURA's positivity constraint is a simplified version of the strictly positivity constraints. When t is $t_1\ t_2$, *ctrs* only appear positively in t if *ctrs* appear positively in t_1 but do not appear in t_2. When t is $(x : t_1) \to t_2$, *ctrs* only appear positively in t if *ctrs* appear positively in t_2 but do not appear in t_1.

Judgment *wf_dom*(*defns*) checks that the type constructors and the data constructors are uniquely declared in *defns*. Finally, we define a function *get_tctr_defns*(*defns*) that strips off the data constructor declarations and returns only the type constructor definitions in *defns*.

The main judgments in checking the well-formedness of signatures are listed below.

Well-formed signatures	$S \vdash \diamond$
Well-formed definitions	$P; S_1; S_2 \vdash defns$
Well-formed type constructors	$S \vdash defns : t$
Well-formed data constructors	$P; S_1; S_2; ctr; t \vdash cdecls$

Note the first two judgments require two signatures. This is because a datatype's declaration may mention constructors defined in the same bundle (e.g. *defns*) by mutual recursion. Such a declaration is checked under a provisional assumption that the rest of its bundle is well-formed. One signature, S_1, is extended with only new type constructors. The other, S_2, is extended with the datatype's entire bundle. The careful separation of S_1 and S_2 allows us to prove decidability of type checking by induction on the structure of S_1, while adding—via S_2—necessary provisional assumptions. We return to this point in Appendix B.

Judgment $P; S_1; S_2; ctr; t \vdash cdecls$ checks that the data constructors *cdecls* defined for *ctr* are well-formed. The signatures S_1 and S_2 are as explained above. *P* is a set of type constructors that can only appear positively in the types in *cdecls*. The type t is the type of *ctr*.

Judgment $P; S_1; S_2 \vdash defns$ checks that a definition is well-formed. S_1, S_2, and *P* have the same meaning as above. Judgment $S \vdash defns : t$ checks the well-formedness of the types given to the type constructors in *defns*. Finally, $S \vdash \diamond$ is the top level judgment for signature well-formedness.

A summary of the rules for type checking signatures is presented in Figure 7. Judgment $S \vdash \diamond$ is recursively defined over the structure of *S*. The rule WF-ASSERT applies when the signature's last bundle an assertion. It checks that the assertion's type constructs a Prop and is classified by Kind. The rule WF-DEFN-TYPE applies when the bundle under scrutiny is composed of datatype definitions in universe Type. It checks that the declared constructors are unique, and that the definitions in *defns* are well-formed in the current signature. The WF-DEFN-PROP rule is similar to the WF-DEFN-TYPE rule except that the definitions are in the Prop universe and occurrences of new type constructors are subject to a positivity constraint. Both the WF-DEFN-TYPE and WF-DEFN-PROP rules call an auxiliary judgments using two signatures as described above.

A bundle is checked for well-formedness by separately examining new type constructors and the new data constructors. A bundle's type constructors are analyzed by $S \vdash defns : t$. The straightforward judgments ensures that the type constructors are well-formed and construct types of the proper kind.

A bundle's data constructors are analyzed by judgment:
$P; S_1; S_2; ctr; t \vdash cdecls$.

The rule WF-CTR-DECLS-CONS checks the main invariants for data constructors. These are:

1. Data constructor declarations do not introduce name conflicts.
2. The data constructor's type, t, is well-formed.
3. t is a curried arrow type with m-many arguments.
4. t's first n arguments (note $n \leq m$) instantiate type datatype's (e.g. *ctr*'s) parameters.
5. t obeys the positivity constraint relative to the names in P. For declarations in Type, P will be empty. For non-trivial declarations in Prop, P will be non-empty.

B. Summary of Typing Rules

Environment typing rules The typing rules for environments are in Figure 8. The first two rules are standard. The last rule WF-ENV-CONS-EQ ensures that an equality binding in the environment is well-formed. AURA allows equality tests between two values of

$$\boxed{P; S_1; S_2; \mathit{ctr}; t \vdash \mathit{cdecls}}$$

$$\frac{}{P; S_1; S_2; \mathit{ctr}; k \vdash \cdot}\ \textsc{Wf-ctr-decls-nil}$$

$$\frac{\begin{array}{c} P; S_1; S_2; \mathit{ctr}; k \vdash \mathit{cdecls} \qquad (c,n) \notin \mathit{dom}(\mathit{cdecls}) \qquad S_1; S_2; \cdot \vdash t : T \\ t = x_1 : s_1 \to x_2 : s_2 \cdots \to x_m : s_m \to (\mathit{ctr}\ x_1 \cdots x_n) \\ k = k_1 \to \cdots \to k_n \to K \text{ where } K = \mathsf{Type} \text{ or } \mathsf{Prop} \\ m \geq n \qquad \mathit{positive}\ P\ t \end{array}}{P; S_1; S_2; \mathit{ctr}; k \vdash \mathit{cdecls} \mid (c,n) : t}\ \textsc{Wf-ctr-decls-cons}$$

$$\boxed{P; S_1; S_2 \vdash \mathit{defns}}$$

$$\frac{}{P; S_1; S_2 \vdash \cdot}\ \textsc{Wf-bundle-ctr-nil} \qquad \frac{P; S_1; S_2 \vdash \mathit{defns} \quad P; S_1; S_2; \mathit{ctr}; t \vdash \mathit{cdecls}}{P; S_1; S_2 \vdash (\mathit{defns} \text{ with data } \mathit{ctr} : t\ \{\mathit{cdecls}\})}\ \textsc{Wf-bundle-ctr-cons}$$

$$\boxed{S \vdash \mathit{defns} : t}$$

$$\frac{S; S; \cdot \vdash t : \mathsf{Kind}}{S \vdash \cdot : t}\ \textsc{Wf-tctr-nil} \qquad \frac{S \vdash \mathit{defns} : k \quad S; S; \cdot \vdash t : \mathsf{Kind} \quad t = (x_1 : t_1) \to \cdots (x_n : t_n) \to k}{S \vdash (\mathit{defns} \text{ with data } \mathit{ctr} : t\ \{\mathit{cdecls}\}) : k}\ \textsc{Wf-tctr-cons}$$

$$\boxed{S \vdash \diamond}$$

$$\frac{}{\cdot \vdash \diamond}\ \textsc{Wf-sig-nil} \qquad \frac{S \vdash \diamond \quad S; \cdot \vdash t : \mathsf{Kind} \quad t = (x_1 : t_1) \to \cdots (x_n : t_n) \to \mathsf{Prop} \quad \mathit{ctr} \notin \mathit{dom}(S)}{S, \mathsf{assert}\ \mathit{ctr} : t \vdash \diamond}\ \textsc{Wf-assert}$$

$$\frac{\begin{array}{c} S \vdash \diamond \qquad S \vdash \mathit{defns} : \mathsf{Type} \\ \mathit{wf\_dom}(\mathit{defns}) \qquad \mathit{dom}(\mathit{defns}) \cap \mathit{dom}(S) = \emptyset \\ \cdot; (S, \mathit{get\_tctr\_defns}(\mathit{defns})); (S, \mathit{defns}) \vdash \mathit{defns} \end{array}}{S, \mathit{defns} \vdash \diamond}\ \textsc{Wf-defn-Type}$$

$$\frac{\begin{array}{c} S \vdash \diamond \qquad S \vdash \mathit{defns} : \mathsf{Prop} \\ \mathit{wf\_dom}(\mathit{defns}) \qquad \mathit{dom}(\mathit{defns}) \cap \mathit{dom}(S) = \emptyset \\ \mathit{dom}(\mathit{defns}); (S, \mathit{get\_tctr\_defns}(\mathit{defns})); (S, \mathit{defns}) \vdash \mathit{defns} \end{array}}{S, \mathit{defns} \vdash \diamond}\ \textsc{Wf-defn-Prop}$$

Figure 7. AURA signature typing rules

atomic types; therefore, t_1 and t_2 have an atomic type k, and k is classified by Type. Since there is no β equivalence at the type level, t_1 and t_2 both have to be values.

$$\boxed{S_1; S_2 \vdash E}$$

$$\frac{}{S_1; S_2 \vdash \cdot}\ \textsc{Wf-env-nil}$$

$$\frac{S_1; S_2 \vdash E \quad S_1; S_2; E \vdash t : k \quad x \text{ fresh}}{S_1; S_2 \vdash E, x{:}t}\ \textsc{Wf-env-cons-var}$$

$$\frac{\begin{array}{c} S_1; S_2 \vdash E \quad S_1; S_2; E \vdash t_1 : k \\ S_1; S_2; E \vdash t_2 : k \quad \mathit{atomic}\ S_2\ k \\ S_1; S_2; E \vdash k : \mathsf{Type} \\ \mathit{val}(t_1) \quad \mathit{val}(t_2) \quad x \text{ fresh} \end{array}}{S_1; S_2 \vdash E, x \sim (t_1 = t_2){:}k}\ \textsc{Wf-env-cons-eq}$$

Figure 8. AURA environment typing rules

Term typing rules We summarize the term typing rules in Figure 9. As we mentioned earlier, the typing judgment for terms needs to take two signature arguments. The typing rules for terms presented in Section 3 is a simplified version and only takes one signature argument. However, most of the rules in Figure 1 become the same as the ones in Figure 9 if the single signature is replaced by the two signatures S_1 and S_2. The only interesting differences are in the WF-TM-CTR, WF-TM-MATCHES and WF-TM-IF rules where one of the two signatures has to be picked for looking up the types of the constructors or for looking up the data constructors of a type constructor. The two signatures only differ when checking the types in datatype declarations; and that when they differ, S_1 is always well-formed but does not contain the data constructors definitions for the bundle that is currently being examined, while S_2 contains the complete data type declarations. S_1 is used for looking up the types of constructors, and S_2 is used for operations that need to look up the data constructors in a datatype declaration. Therefore, in the WF-TM-CTR rule, the type of *ctr* is looked up in S_1; in the WF-TM-MATCHES rule, S_2 is used to check branches coverage; and in the WF-TM-IF rule, S_2 is used to perform the check of atomic types.

Pattern matching Lastly we explain the typing rules for pattern matching, which are listed in Figure 10. The judgment for checking branches has the form $S_1; S_2; E; s; \mathit{args} \vdash \mathit{branches} : t$ where s is the type of the term being analyzed, *args* is the list of type parameters in s, and t is the result type of the match. For instance, if s is *List nat*, the *args* is (*nat*). The rule for the above judgments make use of judgment $S_1; S_2; s; \mathit{args}; t_c; t_b; t_r \vdash \diamond$ for checking the type invariants of each branch.

In the judgment $S_1; S_2; s; \mathit{args}; t_c; t_b; t_r \vdash \diamond$, s and *args* have the same meaning as before, t_c is the type of the data constructor being matched against in the branch, i.e. the type of *cons*, t_b is the type of the body of the branches, and t_r is the result of the pat-

$$\frac{S_1;S_2 \vdash E}{S_1;S_2;E \vdash \mathsf{Type} : \mathsf{Kind}}\ \textsc{Wf-tm-type} \qquad \frac{S_1;S_2 \vdash E}{S_1;S_2;E \vdash \mathsf{Prop} : \mathsf{Kind}}\ \textsc{Wf-tm-prop}$$

$$\frac{S_1;S_2 \vdash E \quad S_1(\mathit{ctr}) = t}{S_1;S_2;E \vdash \mathit{ctr} : t}\ \textsc{Wf-tm-ctr} \qquad \frac{S_1;S_2 \vdash E \quad E(x) = t}{S_1;S_2;E \vdash x : t}\ \textsc{Wf-tm-fv}$$

$$\frac{S_1;S_2;E, x\!:\!t_1 \vdash t_2 : k_2 \quad k_2 \in \{\mathsf{Type}, \mathsf{Prop}, \mathsf{Kind}\}}{S_1;S_2;E \vdash (x\!:\!t_1) \to t_2 : k_2}\ \textsc{Wf-tm-arr}$$

$$\frac{S_1;S_2;E \vdash t : k \quad S_1;S_2;E, x\!:\!t \vdash u : k_1 \quad S_1;S_2;E \vdash (x\!:\!u) \to k_1 : k_2 \quad k \in \{\mathsf{Type}, \mathsf{Prop}, \mathsf{Kind}\} \quad k_2 \in \{\mathsf{Type}, \mathsf{Prop}\}}{S_1;S_2;E \vdash \lambda x\!:\!t.\ u : (x\!:\!t) \to k_1}\ \textsc{Wf-tm-abs}$$

$$\frac{S_1;S_2;E \vdash t_1 : (x\!:\!u_2) \to u \quad S_1;S_2;E \vdash t_2 : u_2 \quad \mathit{val}(t_2) \text{ or } x \notin \mathit{fv}(u)}{S_1;S_2;E \vdash t_1\, t_2 : \{x/t_2\}u}\ \textsc{Wf-tm-app}$$

$$\frac{\begin{array}{l} S_1;S_2;E \vdash e : s \quad s = \mathit{ctr}\ a_1\ a_2 \cdots a_n \quad S_1(\mathit{ctr}) = (x_1 : t_1) \to \cdots (x_n : t_n) \to u \\ \mathit{branches\_cover}\ S_2\ \mathit{branches}\ \mathit{ctr} \quad S_1;S_2;E;s;(a_1,\cdots,a_n) \vdash \mathit{branches} : t \\ S_1;S_2;E \vdash s : u \quad S_1;S_2;E \vdash t : u \quad u \in \{\mathsf{Type}, \mathsf{Prop}\} \end{array}}{S_1;S_2;E \vdash \mathsf{match}\ e\ t\ \mathsf{with}\ \{\mathit{branches}\} : t}\ \textsc{Wf-tm-matches}$$

$$\frac{S_1;S_2 \vdash E}{S_1;S_2;E \vdash \mathsf{prin} : \mathsf{Type}}\ \textsc{Wf-tm-prin} \qquad \frac{S_1;S_2 \vdash E}{S_1;S_2;E \vdash \mathsf{self} : \mathsf{prin}}\ \textsc{Wf-tm-self}$$

$$\frac{S_1;S_2;E \vdash a : \mathsf{prin} \quad S_1;S_2;E \vdash P : \mathsf{Prop}}{S_1;S_2;E \vdash a\ \mathsf{says}\ P : \mathsf{Prop}}\ \textsc{Wf-tm-says}$$

$$\frac{S_1;S_2;E \vdash a : \mathsf{prin} \quad \mathit{val}(a) \quad S_1;S_2;E \vdash p : P \quad S_1;S_2;E \vdash P : \mathsf{Prop}}{S_1;S_2;E \vdash \mathsf{return}_s\ a\ p : a\ \mathsf{says}\ P}\ \textsc{Wf-tm-says-ret}$$

$$\frac{S_1;S_2;E \vdash e_1 : a\ \mathsf{says}\ P \quad S_1;S_2;E \vdash e_2 : (x\!:\!P) \to a\ \mathsf{says}\ Q \quad x \notin \mathit{fv}(Q)}{S_1;S_2;E \vdash \mathsf{bind}_s\ e_1\ e_2 : a\ \mathsf{says}\ Q}\ \textsc{Wf-tm-says-bind}$$

$$\frac{S_1;S_2;\cdot \vdash a : \mathsf{prin} \quad S_1;S_2;\cdot \vdash P : \mathsf{Prop}}{S_1;S_2;E \vdash \mathsf{sign}(a, P) : a\ \mathsf{says}\ P}\ \textsc{Wf-tm-sign} \qquad \frac{S_1;S_2;E \vdash P : \mathsf{Prop}}{S_1;S_2;E \vdash \mathsf{say}\ P : \mathsf{pf}\ \mathsf{self}\ \mathsf{says}\ P}\ \textsc{Wf-tm-say}$$

$$\frac{S_1;S_2;E \vdash P : \mathsf{Prop}}{S_1;S_2;E \vdash \mathsf{pf}\ P : \mathsf{Type}}\ \textsc{Wf-tm-Pf} \qquad \frac{S_1;S_2;E \vdash p : P \quad S_1;S_2;E \vdash P : \mathsf{Prop}}{S_1;S_2;E \vdash \mathsf{return}_p\ p : \mathsf{pf}\ P}\ \textsc{Wf-tm-pf-ret}$$

$$\frac{S_1;S_2;E \vdash e_1 : \mathsf{pf}\ P \quad S_1;S_2;E \vdash e_2 : (x\!:\!P) \to \mathsf{pf}\ Q \quad x \notin \mathit{fv}(Q)}{S_1;S_2;E \vdash \mathsf{bind}_p\ e_1\ e_2 : \mathsf{pf}\ Q}\ \textsc{Wf-tm-pf-bind}$$

$$\frac{S_1;S_2;E \vdash v_1 : k \quad S_1;S_2;E \vdash v_2 : k \quad \mathit{atomic}\, S_2\, k \quad \mathit{val}(v_1) \quad \mathit{val}(v_2) \quad S_1;S_2;E, x \sim (v_1 = v_2)\!:\!k \vdash e_1 : t \quad S_1;S_2;E \vdash e_2 : t}{S_1;S_2;E \vdash \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}\ \textsc{Wf-tm-if}$$

$$\frac{S_1;S_2;E \vdash e : s \quad \mathit{converts}\ E\ s\ t}{S_1;S_2;E \vdash \langle e : t \rangle : t}\ \textsc{Wf-tm-cast}$$

Figure 9. AURA typing rules

$$\boxed{S_1;S_2;E;s;\mathit{args} \vdash \mathit{branches} : t}$$

$$\frac{}{S_1;S_2;E;s;\mathit{args} \vdash \cdot : t}$$

$$\frac{\begin{array}{c} S_1;S_2;E;s;\mathit{args} \vdash b : t_r \quad S_1(c) = t_c \\ S_1;S_2;E \vdash \mathit{body} : t_b \\ S_1;S_2;s;\mathit{args};t_c;t_b;t_r \vdash \diamond \end{array}}{S_1;S_2;E;s;\mathit{args} \vdash b \mid c \Rightarrow \mathit{body} : t_r}$$

$$\boxed{S_1;S_2;s;\mathit{args};t_c;t_b;t_r \vdash \diamond}$$

$$\frac{}{S_1;S_2;s;\cdot;s;t;t \vdash \diamond}$$

$$\frac{S_1;S_2;s;\cdot;t;u;k \vdash \diamond}{S_1;S_2;s;\cdot;(x\!:\!t_1) \to t;(x\!:\!t_1) \to u;k \vdash \diamond}$$

$$\frac{S_1;S_2;s;\mathit{args};\{a/x\}t;u;k \vdash \diamond}{S_1;S_2;s;a,\mathit{args};(x\!:\!t_1) \to t;u;k \vdash \diamond}$$

Figure 10. AURA branches typing rules

tern match. We illustrate the rules through the following example branch.

cons $\rightarrow \lambda$*x*: *nat*. λ*xs*:*List nat*. *b*

In this branch, *b* is the branch body. The result of the pattern match is $t_r =$ *nat* and $s =$*List nat*.

$t_c = (x : \mathsf{Type}) \rightarrow (y : x) \rightarrow (z : \textit{List}\ x) \rightarrow \textit{List}\ x$.

$t_b = (x : \textit{nat}) \rightarrow (xs : \textit{List nat}) \rightarrow \textit{nat}$.

Intuitively, the types of the arguments that the branch body takes is directly linked to the argument types of *cons*, and the return type of the branch body should be the same as t_r. In type checking this branch, first we apply t_c to the list of type parameters *args* (the third rule). In doing so, we reveal the arguments that the branch body should take. Then we check that the t_b takes the same arguments as required by t_c (the second rule). In the end, we should reach a state where $s = t_c$, and $t_b = t_r$ (the first rule).