AURA: PROGRAMMING WITH AUTHORIZATION AND AUDIT

Jeffrey A. Vaughan

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial

Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2009

Prof. Steve Zdancewic of Computer and Information Science
Supervisor of Dissertation

Prof. Jianbo Shi of Computer and Information Science
Graduate Group Chair

Dissertation Committee

Prof. Stephanie Weirich of Computer and Information Science, Chair
Prof. Frank Pfenning of Computer Science, Carnegie Mellon University
Prof. Benjamin C. Pierce of Computer and Information Science
Prof. Andre Scedrov of Mathematics and Computer and Information Science

Acknowledgments

I could not have contemplated, never mind completed, writing this dissertation without the help, patience, and understanding of my colleagues, friends, and family.

Thank you first to Steve Zdancewic. As my academic advisor he taught me how think like a computer scientist. His office door was open, and I learned something new every time I walked through it.

I am grateful to my thesis committee, Frank Pfenning, Benjamin C. Pierce, Andre Scedrov and Stephanie Weirich, for their valuable input and feedback on this dissertation.

Over the last five years I've learned that computer science is social endeavor. To my collaborators and colleagues: our reading groups, seminars, late-night writing sessions, and informal conversations formed the heart of my graduate education. A special thanks to Aaron Bohannon, Limin Jia, Andrew Hilton, Justin Kozer, Karl Mazurak, Joseph Schorr, Haakon Ringberg, Luke Zarko, Jianzhou Zhao, and the University of Pennsylvania PL Club.

Computer and Information Science's administrative and business support staff, especially Mike Felker, cut through miles of red tape over the years.

Thank you to the teachers and professors who encouraged and inspired my interest in mathematics, logic, and computer science. Richard Gaudio and James VanHaneghan first introduced me to the beauty and power of mathematical reasoning. As Andrew Myers's student and teaching assistant I learned the difference between computer engineering and computer science.

My friends in Philadelphia, particularly Aaron, Drew, Hanna, Kristin, Maggie, Nick, and Rob, kept me sane throughout graduate school.

My parents, Joanne and Mark, and my sister, Emily, have always been a consistent source of love and support. My newest family members, Carol Wortman, Dennis Wortman, and Joan Linskey, have adopted me as their own.

Most of all, I thank my wife Jenn. She is my best friend, travel companion, cooking partner, academic strategist, animal trainer, and so much more. She encouraged me during the late nights and early mornings of research and writing, and helped me enjoy even the hardest days of graduate school. And, not least of all, Jenn corrected the spelling of “Jeffrey” on this document’s title page. (Writing a thesis is hard.)

ABSTRACT

Aura: Programming with Authorization and Audit

Jeffrey A. Vaughan

Supervisor: Steve Zdancewic

Standard programming models do not provide direct ways of managing secret or untrusted data. This is a problem because programmers must use ad hoc methods to ensure that secrets are not leaked and, conversely, that tainted data is not used to make critical decisions. This dissertation advocates integrating cryptography and language-based analyses in order to build programming environments for declarative information security, in which high-level specifications of confidentiality and integrity constraints are automatically enforced in hostile execution environments.

This dissertation describes Aura, a family of programing languages which integrate functional programming, access control via authorization logic, automatic audit logging, and confidentially via encryption. Aura's programming model marries an expressive, principled way to specify security policies with a practical policy-enforcement methodology that is well suited for auditing access grants and protecting secrets.

Aura security policies are expressed as propositions in an authorization logic. Such logics are suitable for discussing delegation, permission, and other security-relevant concepts. Aura's (dependent) type system cleanly integrates standard data types, like integers, with proofs of authorization-logic propositions; this lets programs manipulate authorization proofs just like ordinary values. In addition, security-relevant implementation details—like the creation of audit trails or the cryptographic representation of language constructs—can be handled automatically with little or no programmer intervention.

Contents

List of Figures

Chapter 1

Introduction

Standard programming models do not provide direct ways of managing secret or untrusted data. This is a problem because programmers must use ad hoc methods to ensure that secrets are not leaked and, conversely, that tainted data is not used to make critical decisions. This dissertation advocates integrating cryptography and language-based analyses in order to build programming environments for *declarative information security*, in which high-level specifications of confidentiality and integrity constraints are automatically enforced in hostile execution environments.

This document describes a programming language, Aura, that marries a security-sensitive type system for policy specification with cryptography for policy enforcement. In the following, I argue that Aura provides an expressive and sound way to specify and enforce declarative-information-security properties. The remainder of this chapter sketches the Aura system at a high level and outlines the contributions of this dissertation.

1.1 Motivation

Aura is designed to enable mutually distrusting principals to share resources and secrets, subject to a potentially complicated set of policy rules.

For instance, imagine the jukebox system depicted in Figure 1.1. Here two principals (Alice and Bob) are connected to each other and to a server that mediates access to a jukebox. The jukebox is assumed to be completely security-oblivious, and the server is responsible for forwarding valid

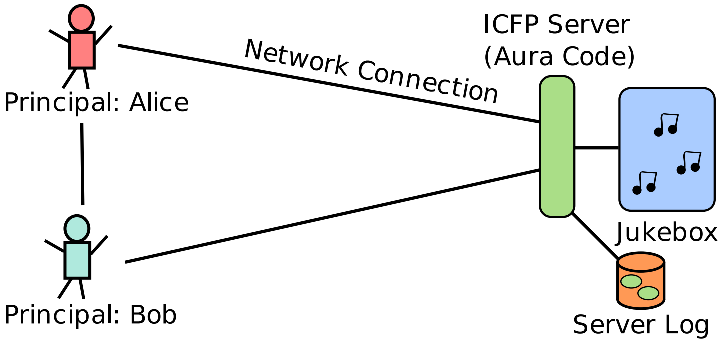


Figure 1.1: Schematic diagram of an Aura jukebox system

requests—and only valid requests—to it. The server itself may be acting on behalf of a principal, such as the RIAA or the *International Cartel for Fonograph Players*, ICFP.

The system might need to enforce a range of policies, and the difficulty of enforcement increases with increasing policy complexity. When ICFP specifies "Alice may play all songs," the server's job is relatively easy. When ICFP specifies "If the Goldbach conjecture holds, then Bob may play songs," the server's job is prohibitively difficult.

Between these extremes, we find more interesting and reasonable policy sets: "ICFP says Alice may play London Calling," "Alice says Bob may play her music," "Bob says Alice may play his new wave songs," and "Bob says Blue Monday is new wave." These policies are interesting for several reasons.

- They contain prominent use of delegation. Sometimes Bob may only play a song because Alice has delegated her rights.

- They are heterogeneous and not necessarily coordinated by a central authority. Alice and Bob might classify songs differently (e.g. "Alice says Blue Monday is pop"), and we will assume that this is a feature of realistic policy sets among potentially mutually distrusting principals.

- Determining access rights is an exercise in distributed decision making. Deciding if Alice is permitted to play a song may require reasoning from several points of view and collecting evidence about these views from diverse sources.

The need for these sorts of policies may be found throughout contemporary computer systems. Users of wikis, distributed files systems, and even electronic door locks employ such access control schemes, or struggle to approximate them when lacking appropriate technological support.

For example, traditional Unix file systems lack a way for unprivileged principals to establish precise file-sharing policies. Files are shared primarily though the creation of groups, an operation that requires administrator intervention. This would not be a problem if users did not wish to share files, or were satisfied with the sharing that can accomplished using a small number of groups. However, a variety of application-specific, access-control regimes, typified by Apache's .htaccess files (Apache, 2009) and techniques for sharing subversion repositories via ssh-tunneling (Collins-Sussman et al., 2008), attest to the fact that users wish to delegate their file-access rights in a fine-grained manner. As in the music example, realistic file-access policies such as "Bob says Alice may update files when subversion agrees," and "Charlie may read .html files under ~/public," feature delegation and lack of central coordination.

Additionally, Bauer et al. (2005a) have demonstrated the utility of decentralized, delegation-rich policies for physical access control. They use computers to check access request for doors, and accommodate expressive policies, such as "Eve says her guests may open her office door," and "Charlie is Eve's guest on January 10th."

Aura builds-on and refines the notion of proof carrying access control (PCA) as a technique to enforce policies like those described above (Appel and Felten, 1999). In Aura, and other PCA systems, processes are required to present, at runtime, proofs that resource accesses conform to policy. In contrast with previous PCA systems, Aura is a single language with fragments tuned for programming and policy definition. Its design ensures that well-typed programs only attempt valid resource accesses. Additionally, Aura's flexible type system can describe a wide variety of security policies. This means that individual applications can use Aura types for PCA in lieu of implementing and debugging a new access control scheme. Such techniques promise to enhance the reliability and security of software systems.

1.2 An Overview of Aura

The Aura family of programming languages integrates functional programming, access control via *authorization logic*, automatic *audit logging*, and *confidentiality types* for protecting secret confidential data. Except when it would cause confusion, we refer to any member the language family—Aura$_0$, Aura, or Aura$_{\text{conf}}$—as Aura.

Evidence-based access control

Aura security policies are expressed as propositions in an authorization logic based on Abadi's (2007) Dependency Core Calculus (DCC). Such logics are suitable for discussing delegation, permission, and other security-relevant concepts. Aura's type system cleanly integrates standard data types (like integers) with proofs of authorization logic propositions, and programs manipulate authorization proofs just as they might other values.

In authorization logics, the proposition A **says** P denotes "Principal A says (or endorses) proposition P." Chapter 2 presents a small-scale model of the Aura language, and describes the role that such endorsed propositions play in authorization and audit. Aura authorization proofs serve as *evidence* of access-control decisions, and programs must present appropriate proofs in order to access resources. Evidence is composed of a mix of cryptographic signatures, which capture principals' "utterances," and standard rules of logical deduction. We argue that automatically logging such evidence enables useful post-hoc analysis of the authorization decisions made during a system's execution, and that log entries should contain proof-evidence documenting access grants. The automated logging of evidence done in Aura provides several compelling benefits. The approach provides a principled way of determining what to log, and logged proofs contain structure that can illuminate policy flaws or misconfigurations. Most importantly, storing unforgeable cryptographic evidence reduces the size and complexity of a system's trusted computing base.

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

Core Aura

Chapter 3 discusses Aura's design in detail. Aura has ML-like evaluation semantics, characterized by call-by-value reduction and effectful operators. Its static semantics are substantially more novel and are based on *weak dependent types* that can express a variety of useful propositions. For instance, the proposition

$$\mathsf{Alice}\ \textbf{says}\ ((\mathsf{P}\colon \textbf{Prop}) \rightarrow \mathsf{Bob}\ \textbf{says}\ \mathsf{P} \rightarrow \mathsf{P})$$

means that principal Alice will endorse any proposition that Bob endorses. This proposition, which may be pronounced "Bob speaks for Alice," means that Alice is delegating *all* of her authority to Bob. In contrast, the proposition

$$\begin{aligned}&\mathsf{Alice}\ \textbf{says}\ ((\mathsf{s}\colon \mathsf{Song}) \rightarrow \mathsf{isJazz}\ \mathsf{s} \rightarrow\\ &\qquad\qquad \mathsf{Bob}\ \textbf{says}\ (\mathsf{MayPlay}\ \mathsf{Bob}\ \mathsf{s}) \rightarrow \mathsf{MayPlay}\ \mathsf{Bob}\ \mathsf{s})\end{aligned}$$

describes a more limited form of delegation, where Alice delegates some, but not all, of her authority to Bob (only her rights to play jazz songs). The latter proposition follows the principle of least privilege and represents a safer, more secure form of delegation than the former. The ability to express such restricted delegation is an advantage of Aura compared with simpler authorization logics containing only polymorphism.

Weak dependent types are simpler than the full-spectrum dependent types found in Coq (Coq, 2006) or Agda (Norell, 2007), and, consequently, Aura enjoys two useful properties: terms have unique types, and type equality is decidable even in the presence of effects. This is achieved by restricting some forms of application and making the use of conversion explicit. These design choices work harmoniously with Aura's call-by-value evaluation relation; many other reduction strategies are incompatible with weak dependency and would induce metatheoretic complexity, or even unsoundness.

The key technical challenge met in Chapter 3 is the elegant combination of functional programming features—including inductive types, general recursion, and world effects—with authorization logic constructs—particularly the **says** type constructor. Aura's design resolves a variety of tensions inherent in this feature set and admits important properties including syntactic soundness and decidable type-checking. This document focuses on the language design itself and provides only a high-level account of the large-scale Coq proofs that demonstrate Aura's metatheoretic properties.

Confidentiality types in Aura$_{\text{conf}}$

Chapter 4 describes new techniques for creating, manipulating, and accessing confidential data. This methodology is compatible with Aura's computation and audit models, and is realized in the Aura$_{\text{conf}}$ programming language extension. Confidential data and computations are given special types and are automatically encrypted as needed. For instance, the type **int** **for** Alice represents an integer readable only by principal Alice. In this system, any user can build secret computations **for** any other. To unprivileged users, confidential data values and computations are opaque.

The **for** type constructor behaves as an indexed monad; treating confidentiality in this manner is both technically robust and aesthetically aligned with Aura's design. As with **says**-propositions, objects with **for**-types are created using the monadic operators **return** and **bind**. A **run** construct takes, for example, an **int** **for** Alice and returns an **int**. Values with **for**-types are represented by (annotated) ciphertexts. This provides security against adversaries that are outside the system and able to, for example, intercept network traffic.

Aura$_{\text{conf}}$ integrates a novel mix of conventional and new ideas to provide intuitive confidentiality operators backed by encryption. The language contains ciphertexts as first-class values. To enable precise, typed-based analysis of these entities, the typechecker can access statically available private keys and examine ciphertexts at *compile* time. When an appropriate key cannot be found, facts about particular ciphertexts (recorded in the runtime's *fact context*) may be used by the typechecker. Ensuring that this enhanced typing regime satisfies preservation is subtle, and is solved here using ideas from modal type theory (Pfenning and Davies, 2001; Jia and Walker, 2004; Murphy, 2008). Aura$_{\text{conf}}$ programs can dynamically access private keys, and soundness requires effects analysis (Lucassen and Gifford, 1988; Talpin and Jouvelot, 1992) that works in concert with modal types. In addition to syntactic soundness, Aura$_{\text{conf}}$ satisfies a noninterference property that gives one precise, but narrow, characterization of the language's security benefits.

1.3 Aura in the Context of Practical Programming

The bulk of this dissertation presents a theory-oriented account of Aura. This section shows how a full Aura implementation would function as a platform for applications development, identifies the boundaries of this work, and describes how Aura concepts can be applied in other settings.

Toward practical programming with Aura

This section sketches how a production-ready Aura implementation could be used for practical programming. Aura applications are intended to run in open, distributed settings. Toward this end, programmers and system architects will need to take a few specific steps to build applications. These steps are introduced below and explained more thoroughly in the following subsections.

First, the programmer must establish a shared vocabulary for programs to describe policies and express data. This ontology is encoded as a *type signature* made up of inductive type definitions and assertion declarations. For instance the following type signature could support this chapter's jukebox example:

```
(∗ Elements of type Song are built from strings. ∗)
data Song: Type { | new_song: string → Song }

(∗ MayPlay constructs a proposition from a principal and a song. ∗)
assert MayPlay: prin → Song → Prop

(∗ isJazz constructs a proposition from a song. ∗)
assert isJazz: Song → Prop
```

Second, the programmer must identify principals that are mentioned directly in source code. *Key configuration files,* as described below, bind principal objects like Alice or SqlServer to public keys that establish identity. The association between principal objects and keys can be changed before deployment without affecting code or recompilation; this allows programs to be reused in different settings.

Third, programmers can write applications that interoperate—a client and server, for instance—using the shared type signature and principal declarations. At this point programming teams could work independently on client and server code; nothing about Aura's architecture requires further coordination between system components. Of course, producing correct code may require collaboration for conventional design and quality control reasons. Note that it's not necessary to implement complete systems in Aura alone; programs can access foreign functions and data from .Net libraries.

Fourth, to deploy a system, administrators must install and configure application binaries on particular hosts. Unlike standard applications, Aura programs must be directed to a key configuration file that provides final bindings between principal objects and their cryptographic keys. Type signatures can be safely compiled into applications and need not be treated specially during deployment. Finally, while Aura is intended to produce logs automatically in a default location, administrators might choose to override the default at this time.

Type signatures

Aura's programming model assumes all components of a distributed system can communicate using a shared type signature that establishes a vocabulary of datatypes, propositions, and policy assertions. It is critical for security that all hosts understand these objects in the same way.

In Aura we have assumed that system nodes maintain independent copies of the type signature, and processes check that data read from the network is well formed with respect to the local copy. When a broken, out-of-date, or malicious client sends bad data due to an incorrect signature, correct clients will be able to reject the message.

Sophisticated systems for distributed programming, including AliceML (Rossberg et al., 2007) and Acute (Sewell et al., 2007) provide a more comprehensive treatment of distribution in a higher-order, typed setting. These systems each provide a careful account of marshaling data values and mobile code. Additionally, Acute programs can selectively *rebind* a type so it refers to definitions that have been updated or that are site-specific. A similar mechanism for associating version and location metadata with Aura type signatures may enable more precise reasoning about system objects and logged proofs.

Log files

Aura is intended to create high-integrity logs of system actions. A full implementation should produce logs automatically, perhaps in ˜/. aura/logs/appname.txt by default. Administrators or advanced users should be able to override the log location by setting an appropriate flag.

This dissertation does not discuss the interesting issue of tool support for audit. Aura programs will create many log entries during execution and these entries will very often contain digital signatures and other inscrutable binary objects. For this reason a full implementation of Aura should include tools to assist auditors with browsing, querying, and interpreting logs.

Principals and modules in Aura

The notion of principal is key to Aura. Principals are intended to be entities that can make policy statements, such as users, organizations, or trusted computer services. We require that a principal's identity can be unambiguously associated with a public cryptography key. A variety of different mechanisms could bind principal identities with public keys. Suitable choices include X.509 certificates in conjunction with global certificate authorities (Cooper et al., 2008), local name bindings via SPKI/SDSI (Ellison et al., 1999), and file based key-management as found in SSH (Ylonen, 1996). Key configuration files would import these externally meaningful identifiers as Aura values of type **prin**. For instance, the following configuration defines Aura principals Alice and Bob using SPKI/SDSI and SSH identities.

```
(* Hash value identifies a unique SPKI/SDSI certificate linking
 * Alice's name to a public key. *)
alias SDSI prin Alice =
  (name (hash sha1 |SLCgPLFlGTzgUbacUMW8kGTEnUk=|) alice liddell)

(* Filename identifies a public key stored in ~/.ssh/ *)
alias SSH prin Bob = id_rsa_bob.pub
```

Such aliases are only needed when principals are mentioned by name in source code.

Every process must also provide a binding for **self**, as follows:

```
(* Self must bind both public and private keys. *)
alias SSH self = id_rsa_vaughan.pub with id_rsa_vaughan
```

A goal of this design is to free programmers from managing keys directly in code. Indeed, keys are loaded by the runtime using key configuration files and are referenced in code only implicitly, by way of principal values. This design makes it impossible for programs to leak private keys directly. Unfortunately, no methodology can completely rule out key leakage via side channels, such as power or timing.

We anticipate that developers will write and debug Aura programs when some relevant keys are unknown. It may be useful to enable placeholder principal declarations like the following:

```
alias FAKE prin Charlie = ()
```

Principal Charlie would not be bound to a key, and the runtime would use simple symbolic expressions to stand in for Charlie's signatures and ciphertexts. The runtime system should only permit such behavior in debug builds, and release builds of Aura programs should not accept **FAKE** principal declarations.

Programs may interact with unknown as well as known principals. For instance, a middleware service might communicate with a distinguished database server represented by principal SqlServer and also process requests from anonymous clients. Aura uses public keys to represent both known (aliased) and unknown (anonymous) principals at runtime. Principals are first-class and can be treated like any other dynamically computed value. In particular, anonymous clients need not appear in the key configuration file. Programmatically, anonymous clients are often handled by packaging principals in inductive datatypes. For instance, Chapter 4 discusses an example program that accepts requests of the following type.

```
data request: Type {
| r_get: prin → Nat → request
| r_put: (a: prin) → Nat → String for a → request
}
```

Both put and get requests are represented by data constructors that take a principal as their first argument. Importantly, request objects may range over any principal—even those unknown at compile or link time. Aura allows comparisons on principals, and it possible to recognize and to specially treat distinguished principals. For instance, the following code skeleton processes a request, giving special consideration to a particular, very important, principal.

```
match r with request {
| r_get → λa: prin. λn: Nat → if a = VIP then ... else ...
| r_put → ...
}
```

Programs run on behalf of principals. Technically, program p runs with Alice's authority when p can use Alice's private key to sign propositions or decrypt data. We anticipate using a module system to capture a program's authority requirements. For instance, the following module uses **say** to sign a proposition with Alice's key. The annotation on its first line indicates that it requires Alice's authority.

```
Module Music Of Alice
  let stringBeanJean: Song = ...

  let share: (s: Song) → Bob says (MayPlay Bob s) →
                              Alice says (MayPlay Bob s) = say (...)
End Module
```

In contrast, the following defines a module that runs with any authority:

```
Module Of ∗
  let affirmTwo: (P: Prop) → (Q: Prop) → self says (And P Q) = ...
End Module
```

Language interoperability

Aura is specialized for writing programs that deal with authorization decisions and confidential data. For this reason, it contains many features unfamiliar to most programmers and lacks others, such as a large standard library, that are common in industrial software platforms. Aura is implemented using the .Net framework, and can interoperate closely with .Net languages such as C♯ or F♯. Mixed-language development provides an appealing strategy that combines Aura's security features with mainstream languages and libraries.

Aura can easily import objects and procedures from foreign languages. This enables a software architecture in which a top-level Aura program interacts with the network, while using .Net modules for major computational work and to interface with resources not protected by Aura policies.

Aura's security guarantees are based the language's type-safety property. However, because Aura's type system is not directly comparable with .Net's, care must be taken to ensure that Aura objects touched by .Net procedures are well-formed. One reasonable programming discipline only passes .Net-native types to .Net procedures. This prevents .Net code from storing references to Aura objects and mangling them at unpredictable times. Additional runtime type checking may enable a more liberal programming style, albeit with some overhead cost.

An inverted architecture, where a top level .Net language calls Aura libraries for proof search, is possible in principle, but we have not thoroughly explored this design space. Static typing for programs built in this style would require either encoding Aura types within .Net interfaces—an interesting research topic—or rewriting the Aura library to expose a simplified type signature. Alternatively, a limited form of dynamic typing may ameliorate the effects of the impedance mismatch. It is unclear how usable the Aura security model would be for clients that cannot understand Aura types.

More critically, there are some fundamentally troubling issues surrounding the use of Aura with .Net languages. The main difficulty is that .Net does not, out of the box, provide a complete mechanism for enforcing abstraction boundaries. For instance, .Net's reflection mechanism can easily query and update the private fields of an abstract datatype.[1] While client code can block particular uses of reflection using .Net's security infrastructure, we are not aware implementation techniques that would affirmatively protect Aura invariants in all contexts. Washburn (2007) studied this problem in the context of functional programming.

This impacts Aura in at least the following specific ways. Aura's type system is too complicated for a shallow embedding in .Net types. Instead, Aura proofs and propositions must handled as a deep embedding. That is, proofs and propositions must be represented by .Net objects. .Net contains both reflection and concurrency. A malicious client may not only damage proofs by tampering with their internal structure, but can also exploit time-of-check-to-time-of-use attacks to subvert local attempts to validate purported proofs. For this reason Aura's end-to-end security properties (such as noninterference, see Chapter 4) are weakened in a mixed-language setting.

Yet all hope is not lost.

First, further research may reduce the magnitude of these problems. Higher-assurance proof handling could be realized with mechanisms—such as moving proof checking to a separate process—that protect proof objects from tampering. It would also be interesting to identify a subset of, for instance, C♯ that can be safely mixed with Aura.

[1] This is accomplished using methods Type.GetFields() and FieldInfo.SetValue() from the System.Reflection namespace. See Microsoft's documentation, `http://msdn.microsoft.com/en-us/library/system.reflection.aspx`, for details.

Second, the problems that occur when mixing .Net and Aura appear at the edges of .Net. In a pedantic sense OCaml is not typesafe due to the standard library function

$$\mathsf{Obj.magic} : {}'\mathsf{a} \to {}'\mathsf{b}$$

which performs an unsafe cast. And Haskell has no reasonable evaluation semantics due to

$$\mathsf{unsafePerformIO} :: \mathsf{IO}\ \mathsf{a} \to \mathsf{a}.$$

Despite these issues, it is widely accepted that programming in OCaml or Haskell is, in an informal sense, safer than programming in C. Similarly, although specific—nearly pathological—uses of .Net may undermine parts of Aura, .Net programmers could see a real, practical benefit from mixing Aura into their applications.

As it stands, we claim that Aura mixed with .Net provides a substantially more secure platform than .Net alone, and a more immediately practical platform than Aura alone.

Scope of Aura

This dissertation's primary goal is to investigate type-based approaches to access control, audit, and confidentiality. Toward this end, the languages discussed in the thesis—Aura$_0$, Aura, and Aura$_{\text{conf}}$—are defined as intermediate languages. To simplify definition, compilation, and analysis, they omit less relevant surface-language features including type inference, exceptions, and general mutable state. Chapter 5 does, however, briefly address some of these issues. Likewise a full treatment of a modern module system (Dreyer and Rossberg, 2008) or tighter integration with .Net (Syme et al., 2007) is outside the scope of this work.

We adopt the stance that the programmer should be able to use Aura without directly thinking about authentication, cryptography, or logging mechanisms. Primitive operations such as nonce generation and explicit encryption are therefore excluded from the Aura programming model. Several researchers have investigated programming languages for low-level protocol definition, notably Bengtson et al. (2008). As mentioned above, Aura explicitly does not discuss the related issues of establishing and maintaining a public key infrastructure (Ellison and Schneier, 2000). Additionally, we do not investigate mechanisms for ensuring the integrity of audit logs; this has been treated elsewhere in the literature (Bellare and Yee, 1997; Schneier and Kelsey, 1998).

Technology transfer

Aura brings to the table several general concepts useful for building pragmatic secure systems.

Systematically auditing *why* access control decisions are made is important. Because fully predicting the consequences of access-control policy changes is, in nontrivial systems, undecidable (Harrison et al., 1976), reference monitors may always make some surprising decisions. Chapter 2 demonstrates that after-the-fact audit of access grants can be used to understand, refine, and improve security policy. Furthermore, recording why a grant occurred enables policies to include escape hatches that permit resource access in emergency situations. Such escape hatches are valuable when the expected cost of false denies exceeds the cost of false allows.

Access-control policies should be defined in flexible, expressive logics. Aura's access-control logic is based on constructive type theory, and, as shown in examples throughout the dissertation, this provides a framework for defining and reasoning about fine-grained polices. Similar designs have been used to mediate access to file systems (Garg and Pfenning, 2009) and physical doors (Bauer et al., 2005a). As argued above, conventional mechanisms, like .htaccess files, are less able to express desired policies, compared with logic-based mechanisms. This informal observation has been affirmed when comparing access-control logics with physical keys (Bauer et al., 2008).

Aura's model of secure programming is based on authentic credentials and explicitly confidential data values. In contrast, many standard security models focus on the integrity and confidentiality of channels or locations, such as network connections (Ylonen, 1996) or program variables (Volpano et al., 1996). An advantage of the Aura approach is that confidential data can be written to disk or the network without requiring that these output devices satisfy particular security constraints. Likewise, proofs serving as authentic credentials can be trusted even when they are delivered by low integrity methods, such as http.

1.4 Contributions

The principle contributions of this dissertation are as follows.

- The design of core Aura, a language with support for first-class, dependent authorization policies. Aura's design includes weak dependency, unique typing of expressions, and data-centric access control via the **says** modality.

- The main metatheory results for Aura: syntactic soundness and decidable typechecking. Full proofs of these properties are mechanized in Coq. (As noted below, these proofs are the result of collaboration.)

- The design and description of Aura's audit model and related system architecture. Aura uses a small, trusted interface to realize access control policies specified by mutually distrustful principals using Aura's **says** construct.

- Examples and discussion showing that proof-theoretic properties, such as subject reduction and normalization, can play a useful role in audit and that this architecture minimizes the system's trusted computing base.

- The design of $\text{Aura}_{\text{conf}}$ including the confidentiality type constructor **for** and a sophisticated type system that enforces both key-management and code-mobility constraints.

- A mechanized proof that $\text{Aura}_{\text{conf}}$ is syntactically sound and satisfies a noninterference property.

1.5 Bibliographic Notes

The work described in this document would not have been possible with out the help and hard work of my collaborators. Chapter 2 reports on joint work with Limin Jia, Karl Mazurak, and Steve Zdancewic that was originally published at CSF (Vaughan et al., 2008). Chapter 3 describes research conducted with Limin Jia, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. The original paper was published at ICFP (Jia et al., 2008). Long versions of these papers are available as University of Pennsylvania technical reports numbers MS-CIS-08-09 and MS-CIS-08-10. Although I initiated the formal development of the mechanized proofs therein, it was Limin Jia who did the hard work of "getting to QED." Chapter 4 reports on new research.

This research is supported in part by NSF grants CNS-0346939, CCF-0524035, CNS-0524059, and CCF-0716469.

Chapter 2

Aura: Programming with Audit in Aura

This chapter describes Aura's setting and introduces a simplified version of the language, Aura$_0$, that is useful for discussing authorization and audit. We demonstrate that Aura$_0$'s propositional fragment is a consistent logic, via proofs of syntactic soundness and strong normalization, and describe how Aura$_0$'s (and by extension Aura's) use of proof-carrying authorization facilitates the design of secure software.

This chapter's results are joint work with Limin Jia, Karl Mazurak, and Steve Zdancewic.

2.1 Introduction

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

Despite the practical importance of auditing, there has been surprisingly little research into what constitutes good auditing procedures.[1] There has been work on cryptographically protecting logs to prevent or detect log tampering (Schneier and Kelsey, 1998; Bellare and Yee, 1997), efficiently searching confidential logs (Waters et al., 2004), and experimental research on effective, practical logging (Axelsson et al., 1998; Ko et al., 1997). But there is relatively little work on *what* the contents of an audit log should be or how to ensure that a system implementation performs appropriate logging (see Wee's (1995) paper on a logging and auditing file system for one approach to these issues, however).

In this chapter, we argue that audit log entries should constitute *evidence* that justifies the authorization decisions made during the system's execution. Following an abundance of prior work on authorization logic (Abadi et al., 1993; Jajodia et al., 1997; DeTreville, 2002; Abadi, 2003; Li et al., 2003; Abadi, 2006; Garg and Pfenning, 2006; DeYoung et al., 2008) we adopt the stance that log entries should contain *proofs* that access should be granted. Indeed, the idea of logging such proofs is implicit in the proof-carrying authorization literature (Appel and Felten, 1999; Bauer, 2003; Bauer et al., 2005b), but, to our knowledge, the use of proofs for auditing purposes has not been studied directly.

There are several compelling reasons why it is advantageous to include proofs of authorization decisions in the log. First, by connecting the contents of log entries directly to the authorization policy (as expressed by a collection of rules stated in terms of the authorization logic), we obtain a principled way of determining what information to log. Second, proofs contain structure that can potentially help administrators find flaws or misconfigurations in the authorization policy. Third, storing verifiable evidence helps reduce the size of the trusted computing base; if every access-restricting function automatically logs its arguments and result, the reasoning behind any particular grant of access cannot be obscured by a careless or malicious programmer.

The impetus for this chapter is our experience with the design and implementation of Aura. The primary goal of this work is to find mechanisms that can be used to simplify the task of manipulating authorization proofs and to ensure that appropriate logging is always performed regardless of how a reference monitor is implemented. Among other features intended to make building secure software

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

easier, Aura provides a built-in notion of principals, and its type system treats authorization proofs as first-class objects; the authorization policies may themselves depend on program values.

This chapter focuses on the use of proofs for logging purposes and the way in which we envision structuring Aura software to take advantage of the authorization policies to minimize the size of the trusted computing base. The contributions of this chapter can be summarized as follows.

Section 2.2 proposes a system architecture in which logging operations are performed by a trusted kernel, which can be thought of as part of the Aura runtime system. Such a kernel accepts proof objects constructed by programs written in Aura and logs them while performing security-relevant operations.

To illustrate Aura more concretely, Section 2.3 develops a dependently typed authorization logic based on DCC (Abadi, 2006) and similar to that found in the work by Fournet, Gordon, and Maffeis (2005, 2007). This language, Aura$_0$, is intended to model the fragment of Aura relevant to auditing. We show how proof-theoretic properties such as subject reduction and normalization can play a useful role in this context. Of particular note is the normalization result for Aura$_0$ authorization proofs.

Section 2.4 presents an extended example of a file system interface; as long as a client cannot circumvent this interface, any reference monitor code is guaranteed to provide appropriate logging information. This example also demonstrates how additional domain-specific rules can be built on top of the general kernel interface, and how the logging of proofs can be useful when it isn't obvious which of these rules are appropriate.

Of course, there are many additional engineering problems that must be overcome before proof-enriched auditing becomes practical. Although it is not our intent to address all of those issues here, Section 2.5 highlights some of the salient challenges and sketches future research directions.

2.2 Kernel Mediated Access Control

A common system design idiom protects a resource with a *reference monitor*, which takes requests from (generally) untrusted clients and decides whether to allow or deny access to the resource (Bishop, 2002). Ideally a reference monitor should be configured using a well-specified set of *rules* that define the current access-control policy and mirror the intent of some institutional policy.

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

1. The reference monitor implementation or rule language may be insufficient to express institutional intent. It this case, the rules must necessarily be too restrictive or too permissive.

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

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

The first and second points illustrate an interesting tension: rule language expressiveness is both necessary and problematic. While overly simple languages prevent effective realization of policy intent, expressive languages make it more likely that a particular rule set has unintended consequences. The latter issue is particularly acute in light of Harrison and colleagues' observation that determining the ultimate effect of policy changes—even in simple systems—is generally undecidable (Harrison et al., 1976). The third point recognizes that reference monitors may be complex and consequently vulnerable to implementation flaws.

The Aura programming model suggests a different approach to protecting resources, illustrated in Figure 2.1. There are three main components in the system: a trusted kernel, an untrusted application, and a set of rules that constitute the formal policy. The kernel itself contains a log and a resource to be protected. The application may only request resource access through kernel interface I_K. This interface (made up of the op_is in the figure) wraps each of the resource's native operations (the $\mathsf{raw\text{-}op}_i$s) with new operations taking an additional argument—a proof that access is permitted. Σ_K and Σ_{ext} contain constant predicate symbols that may be occur in these proofs.

Unlike the standard reference monitor model, an Aura kernel forwards every well-typed request to its underlying resource. Each op_i function takes as an additional argument a proof that the operation is permitted and returns a corresponding proof that the operation was performed, so the well-typedness of a call ensures that the requested access is permitted. Proofs can be typechecked dynamically in time linearly proportional to the size of the proof should the request not come from

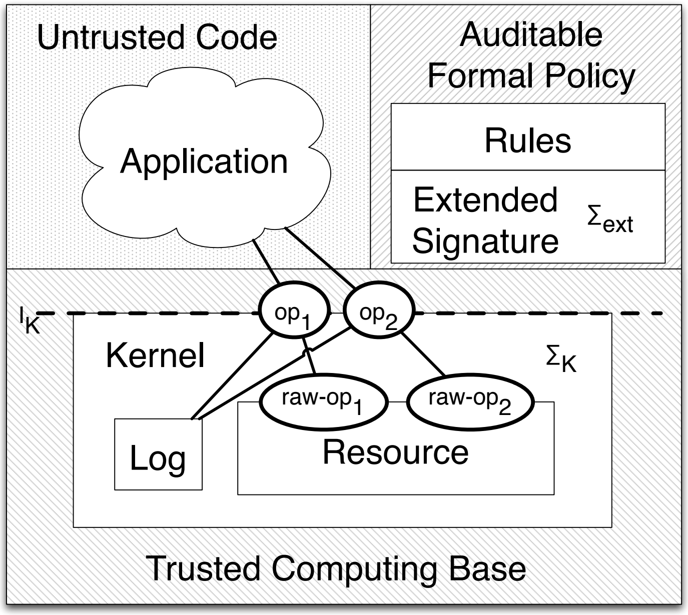


Figure 2.1: Schematic of the Aura runtime environment

a well-typed application. Moreover, logging these proofs is enough to allow an auditor to ascertain precisely why any particular access was allowed.

We define a language Aura$_0$ to provide an expressive policy logic for writing rules and kernel interface types. It is a cut-down version of full Aura, which itself is a polymorphic and dependent variant of Abadi's Dependency Core Calculus (Abadi et al., 1999; Abadi, 2006). In Aura$_0$, software components may be explicitly associated with one or more principals. Typically, a trusted kernel is identified with principal K, and an untrusted application may work on behalf of several principals: A, B, etc. Principals can make assertions; for instance, the (inadvisable) rule "the kernel asserts that all principals may open any file," is written as proposition K **says** $((A{:}\textbf{prin}) \rightarrow (f{:}\textbf{string}) \rightarrow \mathsf{OkToOpen}\ A\ f)$. Evidence for this rule contains one or more signature objects—possibly implemented as cryptographic signatures—that irrefutably tie principals to their utterances.

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

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

The formal system description

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

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

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

The current state σ is an arbitrary structure representing R's current state, and *States* is the set of all possible resource states. I_R is the resource's interface; each $\textsf{raw-op}_i : T_i \;\Rightarrow\; S_i$ is an operator with its corresponding type signature. Type $T_i \;\Rightarrow\; S_i$ indicates that the operation takes an element of T_i, returns an element of S_i, and may have side effects; types will be explained in detail later. The transition function δ describes how the raw operations update state, as well as their input-output behavior. For instance, $(u, \sigma') = \delta(\textsf{raw-op}_i, v, \sigma)$ when raw operation i—given input v and initial resource state σ—produces output u and updates the resource state to σ'.

We formalize a trusted kernel K as a tuple (L, R, Σ_K, I_K); the authority of the kernel is denoted by the constant principal $\textsf{K}$. The first component, L, is a list of proofs representing the log. The second component is the resource encapsulated by the kernel. Signature Σ_K contains pairs of predicates, $\textsf{OkToOp}_i : T_i \rightarrow \textbf{\textsf{Prop}}$ and $\textsf{DidOp}_i : T_i \rightarrow S_i \rightarrow \textbf{\textsf{Prop}}$ for each $\textsf{raw-op}_i$ of type $T_i \;\Rightarrow\; S_i$ in I_R. (The notation $P : T \rightarrow \textbf{\textsf{Prop}}$ declares that P is a unary operator which builds propositions, and

$P : T \to S \to \textsf{\textbf{Prop}}$ declares a binary proposition constructor; see Section 2.3 for details.) These predicates serve as the core lexicon for composing access control rules: a proof of $\textsf{K}\ \textsf{\textbf{says}}\ \textsf{OkToOp}\ t$ signifies that an operation raw-op is permitted with input t, and a proof of $\textsf{K}\ \textsf{\textbf{says}}\ \textsf{DidOp}\ t\ s$ means that raw-op was run with input t and returned s. Associating t and s with this proof lets system auditors establish how some values were computed, not just what events occurred. Lastly, the kernel exposes an application programming interface I_K, containing a security-aware wrapper operation

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

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

The type of $\textsf{op}_i$ shows that the kernel requires two arguments before it will provide access to $\textsf{raw-op}_i$. The first argument is simply $\textsf{raw-op}_i$'s input; the second is a proof that the kernel approves the operation, typically a composition of policy rules (globally known statements signed by K) and statements made by other relevant principals. The return value of $\textsf{op}_i$ is a pair of $\textsf{raw-op}_i$'s output with a proof that acts as a receipt, affirming that the kernel called $\textsf{raw-op}_i$ and linking the call's input and output. Note that $\textsf{OkToOp}_i$ and $\textsf{DidOp}_i$ depend on the arguments x and y.

The final components in the model are the application, the rule set, and the extended signature. We assume either that the application is well-typed—and thus that it respects I_K—or, equivalently, that the kernel performs dynamic typechecking (i.e., proof-checking) on incoming untrusted arguments. The rule set is simply a well-known set of proofs intended to represent some access control policy. As we will see, proofs in the rule set may contain interesting internal structure. The extended signature (Σ_{ext} in Figure 2.1) defines predicate symbols that these rules may use in addition to those defined in Σ_K.

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

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

and the kernel interface

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

A trivial policy could allow remote procedure calls on all strings. This policy may be realized by the singleton rule set

$$\mathit{Rules} = \{r_0 : \mathsf{K}\ \textbf{says}\ ((x{:}\textbf{string}) \rightarrow \mathsf{OkToRPC}\ x)\}.$$

State transition semantics

While the formalism presented thus far is sufficient to describe what Aura$_0$ systems look like at one instant in time, it is much more interesting to consider an evolving system. Here we semi-formally describe three alternative operational semantics for Aura$_0$ systems, making comparisons based on audit capability.

To demonstrate the key components of authorization and auditing in Aura$_0$, we consider evaluation from the three perspectives described below. Each updates state according to the transition relations defined in Figure 2.2.

1. Resource evaluation, written with $-\{\}\!\!\rightarrow^r$, models the state transition for raw resources. This relation does no logging and does not consider access control.

2. Logged evaluation, written with $-\{\}\!\!\rightarrow^l$, models state transitions of an Aura$_0$ system implementing logging as described in this paper. All proofs produced or consumed by the kernel are recorded in the log.

3. Semi-logged evaluation, written with $-\{\}\!\!\rightarrow^s$, models the full system update with weaker logging. While proofs are still required for access control, the log contains only operation names, not the associated proofs.

Resource evaluation is the simplest evaluation system. A transition $I_R; \delta \vdash \sigma -\{\mathsf{raw\text{-}op}_i v\}\!\!\rightarrow^r \sigma'$ may occur when v is a well typed input for $\mathsf{raw\text{-}op}_i$ according to resource interface I_R and δ specifies that $\mathsf{raw\text{-}op}_i$, given v and starting with a resource in state σ, returns some result u and updates the

$$\boxed{\text{Resource evaluation relation} \quad I_R; \delta \vdash \sigma \mathrel{-\{o\}\!\!\rightarrow^{r}} \sigma'}$$

$$\frac{\cdot;\cdot \vdash v : T \qquad \textsf{raw-op}_i : T \Rightarrow S \in I_R \qquad (u, \sigma') = \delta(\textsf{raw-op}_i, v, \sigma)}{I_R; \delta \vdash \sigma \mathrel{-\{\textsf{raw-op}_i\ v\}\!\!\rightarrow^{r}} \sigma'}\ \textsc{r-Act}$$

$$\boxed{\text{Semi-logged evaluation relation} \quad \Sigma_{ext}; I_{\mathsf{K}}; \delta \vdash \sigma \mathrel{-\{o\}\!\!\rightarrow^{s}} \sigma'}$$

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

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

$$\boxed{\text{Proof-logged evaluation relation} \quad \Sigma_{ext}; I_{\mathsf{K}}; \delta \vdash \sigma \mathrel{-\{o\}\!\!\rightarrow^{l}} \sigma'}$$

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

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

Figure 2.2: Operational semantics

resource state to σ'. (In the following we will generally omit the $\vdash$ and objects to its left, as they are constant and can be inferred from context.)

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

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

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

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

By examining the rules in Figure 2.2, we can see that the kernel may only sign DidOp receipts during evaluation. Since statements signed by any other principal may be added to $\mathcal{S}$ at any time, we may identify the initial set of **sign** objects in $\mathcal{S}$ with the system's policy rules.

Observe that rules S-SAY and L-SAY do not specify that only principal A can create **sign**(A, P) objects. Though desirable, such a property is not necessary for this chapter's treatment of signing. The language introduced in Chapter 3 enforces such a restriction implicitly, and Chapter 4 provides an explicit and precise formalism for understanding a program's runtime authority. In Chapter 4 a program is run with a private key specifying a principal on whose authority it acts, and a program can only generate signatures with an appropriate private key. In contrast, S-SAY and L-SAY record that a new signature was created—but do not address *how* or *where* this occurred.

Audit and access control The three transition relations permit different operations and record different information about allowed actions. Resource evaluation allows all well-typed calls to the raw interface, and provides no information to auditors. Semi-logged evaluation allows only authorized access to the raw interface via access control, and provides audit information of the list of allowed operations. Logged evaluation, like semi-logged evaluation, allows only authorized access to the raw interface; it also produces a more informative log of the proofs of the authorization decisions. Intuitively, semi-logged and logged evaluation, which deploy access control, allow strictly fewer operations than resource evaluation. Logged evaluation provides more information than the semi-logged evaluation for auditing, and semi-logged evaluation provides more information than resource evaluation.

The rest of this section sketches a technical framework in which the above claims are formalized and verified. The main result, Lemma 1, states that logged evaluation provides more information during audit than resource evaluation; similar results hold when comparing the logged and semi-logged relations or the semi-logged and resource relations. The result is unsurprising, but it is not trivial to make precise. Furthermore, as will be discussed in Section 2.6, surprisingly little work has addressed the formal semantics of logging, and it is worthwhile to verify that this logging model behaves as expected.

The lemma's formal statement requires a few auxiliary concepts.

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

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

Logged and semi-logged traces are defined similarly.

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

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

For a set of traces, $\lfloor (H) \rfloor_{l/r}$ is defined as $\{ \lfloor \tau \rfloor_{l/r} \mid \tau \in H \}$. The l/r subscript indicates erasure from logged to resource traces. Analogous functions, $\lfloor \cdot \rfloor_{l/s}$ and $\lfloor \cdot \rfloor_{s/r}$, can be defined to relate other pairs of evaluation schemes.

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

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

Lemma 1. *There exists a kernel K, extended signature Σ_{ext}, configuration $C = (L, \sigma, \mathcal{S})$, rule set $\mathcal{S}_0$, initial trace σ_0 and resource trace τ such that $\tau \in H^r(\sigma_0, \sigma)$, but $\tau \notin \lfloor(H^l(\sigma_0, \mathcal{S}_0, C))\rfloor_{l/r}$.*

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

Not surprisingly, it is possible to make similar distinctions between logged and semi-logged histories, as logged histories can ensure that a particular L-ACT step occurred, but this is not possible in the semi-logged case. As we will see in Section 2.3, this corresponds to the inability of the semi-logged system to distinguish between different proofs of the same proposition and thus to correctly assign blame.

We implement operations in I_K by wrapping each underlying raw-op with trusted logging code. Each wrapped operation is defined as follows[2]:

```
op$_i$ = $\lambda v.\lambda p.$
  do logAppend($p$)
  let $s$ = raw-op$_i$ $v$
  let $p'$ = sign(K, DidOp$_i$ $v$ $s$)
  do logAppend($p'$)
  return $\langle s, p' \rangle$
```

The function op$_i$ takes two inputs: a term v (to be passed to the underlying raw-op$_i$) and a proof p. It returns a pair $\langle s, p' \rangle$ whose first component the result of applying raw-op$_i$ to v and whose second component is a proof generated by the kernel to provide evidence that the operation was performed.

[2] This example is written in simple imperative pseudocode. Its syntactic similarity to Aura$_0$, as specified in Section 2.3, is not significant.

The proof p (recorded in the log on line 2) must inhabit the type K **says** $\mathsf{OkToOp}_i\ v$. Lines 3 and 4 call into the underlying resource and construct a signature object attesting that this call occurred; line 5 records this newly created proof.

2.3 The Logic

This section defines Aura$_0$, a language for expressing access control. Aura$_0$ is a higher-order, dependently typed, cut-down version of Abadi's Dependency Core Calculus (Abadi et al., 1999; Abadi, 2006), Following the Curry-Howard isomorphism (Curry et al., 1958), Aura$_0$ types correspond to propositions relating to access control, and expressions correspond to proofs of these propositions. Dependent types allow propositions to be parameterized by objects of interest, such as principals or file handles. The interface between application and kernel code is defined using this language.

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

Syntax

Figure 2.3 defines the syntax of Aura$_0$, which features two varieties of terms: access control proofs p, which are classified by corresponding propositions P of kind **Prop**, and conventional expressions e, which are classified by types T of the kind **Type**. For ease of the subsequent presentation of the typing rules, we introduce two sorts, **Kind$^{\mathbf{P}}$** and **Kind$^{\mathbf{T}}$**, which classify **Prop** and **Type** respectively. The base types are **prin**, the type of principals, and **string**; we use x to range over variables, and a to range over constants. String literals are `""`–enclosed ASCII symbols; A, B, C etc. denote literal principals, while principal variables are written A, B, C.

In addition to the standard constructs for the functional dependent type $(x{:}t_1) \rightarrow t_2$, dependent pair type $\{x{:}t_1; t_2\}$, lambda abstraction $\lambda x{:}t_1.\, t_2$, function application $t_1\ t_2$, and pair $\langle t_1, t_2\rangle$, Aura$_0$ includes a special computational function type $(x{:}t_1) \Rightarrow t_2$. Intuitively, $(x{:}t_1) \rightarrow t_2$ is used for logical implication and $(x{:}t_1) \Rightarrow t_2$ describes kernel interfaces; Section 2.3 discusses this

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

Figure 2.3: Syntax of Aura$_0$

further. We will sometimes write $t_1 \rightarrow t_2$, $t_1 \ \Rightarrow \ t_2$, and $\{t_1; t_2\}$ as a shorthand for $(x{:}t_1) \rightarrow \ t_2$, $(x{:}t_1) \ \Rightarrow \ t_2$, and $\{x{:}t_1; t_2\}$, respectively, when x does not appear free in t_2.

As in DCC, the modality **says** associates claims relating to access control with principals. The term $\textbf{return}@[A] \ p$ creates a proof of A **says** P from a proof of P, while **bind** $x \ = \ p_1$ **in** p_2 allows a proof of A **says** P_1 to be used as a proof of P_1, but only within the scope of a proof of A **says** P_2. Finally, expressions of the form $\textbf{sign}(\mathsf{A}, P)$ represent assertions claimed without proof. Such an expression is indisputable evidence that P was asserted by A—rather than, for example, someone to whom A has delegated authority. Such signed assertions must be verifiable, binding (i.e. non-repudiable), and unforgeable; signature implementation strategies are discussed in Section 2.5.

Technically, Aura$_0$'s grammar contains only a single syntactic category. The typing relation is responsible for categorizing each term as an expression, proof, type, proposition, or kind. The grammar in Figure 2.3 is imprecise, but establishes a useful metavariable convention. For instance, e and p indicate expressions and proofs. We will observe this convention in the remainder of this chapter and in Chapters 3 and 4. Purists may read e, p, T, P, and k as t.

Substitution and free variable functions—written $\{\cdot/\cdot\}\cdot$, $fv(\cdot)$, and $dom(\cdot)$—are defined as usual. See Figure 2.4 for details.

Type system

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

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

Typechecking consists of four judgments:

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

The signature Σ is well formed if Σ maps constants to types of sort $\textsf{\textbf{Kind}}^{\mathbf{P}}$—in other words, all Aura$_0$ constants construct propositions. The context Γ is well formed with respect to signature Σ if Γ maps variables to well-formed types. The well-formedness judgments are defined in Figure 2.7. A summary of the typing rules for terms can be found in Figures 2.5 and 2.6. Most of the rules are straightforward, and we explain only a few key rules.

Rule T-SIGN states that **sign**(A, P)—a signed assertion of proposition P by principal A—has type A **says** P. Here P can be any proposition, even false. The judgment $\Sigma \vdash \diamond$ ensures that propositional constants, such as OkToRPC, have no constructors. Thus signed assertions are the only way to create interesting access control statements. T-SIGN typechecks A and P in the empty variable context; **sign** objects are intended to have unambiguous meaning in any scope, and nested free variables are meaningless.

The rule T-RETURN states that if we can construct a proof term p for proposition P, then the term **return**$@[A]\ p$ is a proof term for proposition A **says** P—in other words, all principals believe what can be independently verified. The T-BIND rule is a standard bind rule for monads and, intuitively, allows anyone to reason from A's perspective.

The rule for the functional dependent type T-ARR restricts the kinds of dependencies allowed by the type system, ruling out dependencies on objects of kind **Type**. Note that, in the T-LAM

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

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

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

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

Free variables of term t, $fv(t)$:

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

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

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

Figure 2.4: Substitution and free variable functions are defined as usual.

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

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

$$\frac{\Sigma \vdash \Gamma \qquad T \in \{\textbf{string}, \textbf{prin}\}}{\Sigma; \Gamma \vdash T : \textbf{Type}}\ \textsc{T-base} \qquad\qquad \frac{\Sigma \vdash \Gamma \qquad x : t \in \Gamma}{\Sigma; \Gamma \vdash x : t}\ \textsc{T-var}$$

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

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

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

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

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

$$\frac{\Sigma \vdash \Gamma \qquad A \in \{\mathsf{A}, \mathsf{B} \ldots\}}{\Sigma; \Gamma \vdash A : \textbf{prin}}\ \textsc{T-litPrin} \qquad\qquad \frac{\Sigma \vdash \Gamma \qquad s \in \{\texttt{"a"}, \texttt{"b"}, \ldots\}}{\Sigma; \Gamma \vdash s : \textbf{string}}\ \textsc{T-litStr}$$

$$\frac{\Sigma; \Gamma \vdash e_1 : t\ \textbf{says}\ P_1 \quad \Sigma; \Gamma, x : P_1 \vdash e_2 : t\ \textbf{says}\ P_2 \quad x \notin \mathit{fv}(P_2)}{\Sigma; \Gamma \vdash \textbf{bind}\ x\ =\ e_1\ \textbf{in}\ e_2 : t\ \textbf{says}\ P_2}\ \textsc{T-bind}$$

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

$$\frac{\Sigma; \Gamma \vdash t_1 : (x{:}P_2) \rightarrow\ P \qquad \Sigma; \Gamma \vdash t_2 : P_2}{\Sigma; \Gamma \vdash t_1\ t_2 : \{t_2/x\}P}\ \textsc{T-app}$$

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

Figure 2.5: Expression typing

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

$$\frac{\Sigma; \Gamma \vdash t_1 : t_2 \quad t_2 \in \{\textsf{\textbf{Kind}}^{\mathbf{P}}, \textsf{\textbf{Kind}}^{\mathbf{T}}, \textsf{\textbf{Prop}}, \textsf{\textbf{Type}}\}}{\Sigma; \Gamma \vdash t_1} \text{ T-C}$$

$$\frac{\Sigma; \Gamma \vdash t_1 : k \quad k \in \{\textsf{\textbf{Type}}, \textsf{\textbf{Prop}}\} \quad \Sigma; \Gamma, x : t_1 \vdash t_2}{\Sigma; \Gamma \vdash (x{:}t_1) \ \Rightarrow\ t_2} \text{ T-ARR-C}$$

Figure 2.6: Command typing

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

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

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

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

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

Figure 2.7: Well formed signature and environment judgments (defined mutually with typing relation)

rule, the type of the lambda abstraction must be of kind **Prop**. With such restrictions in place, it is straightforward to observe that these two rules allow us to express flexible access control rules while at the same time ruling out type-level computations and preserving decidability of type checking. These heavy restrictions will be liberalized in subsequent chapters; for now new objects with kind **Type** can only be introduced with constants or pairing.

The interface between application code and the kernel also requires a type description. For this reason, Aura$_0$ introduces a special computational arrow type, $(x{:}t_1) \ \Rightarrow\ t_2$. Computation types are used only to classify top-level operators, such as Section 2.2's rpc command, and computations cannot appear in proofs or propositions. This decouples Aura$_0$ proof reduction from effectful computation and simplifies the interpretation of propositions. There is no Aura$_0$ syntax to introduce new computations. In contrast full Aura (Chapter 3) has only a single variety of arrow; the typing relation prevents computations from occurring in positions that are logically meaningless or unsound.

Note that defining separate computational and logical sorts, $\textbf{Kind}^{\mathbf{T}}$ and $\textbf{Kind}^{\mathbf{P}}$, makes it easy to syntactically distinguish type-level objects. Full Aura defines a very similar type system using only a single sort; this makes some of its typing rules slightly heavier, but the two approaches are largely equivalent.

The typing rule T-PAIRTYPE for dependent pairs is standard and permits objects of kinds **Type** and **Prop** to be freely mixed; for simplicity we prohibit types and propositions themselves from appearing in a pair. Note that Aura$_0$ features an introduction form for pairs but no corresponding elimination form. This simplifies the treatment of Aura$_0$ but is not essential; full Aura includes a **match** construct to eliminate datatypes.

Example

The combination of dependent types and the **says** modality in Aura$_0$ can express many interesting policies. For instance, Abadi's encoding of speaks-for (Abadi, 2006) is easily adopted:

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

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

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

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

Recall also the Remote Procedure Call example from Section 2.2. While an application might use r_0 (of type K **says** $((x{:}\textbf{string}) \rightarrow\; \mathsf{OkToRPC}\; x))$ directly when building proofs, it could also construct a more convenient derived rule by using Aura$_0$'s **bind** to reason from K's perspective. For instance:

$$\begin{array}{rcl} r_0' & : & (x{:}\textbf{string}) \rightarrow\; \mathsf{K} \;\textbf{says}\; \mathsf{OkToRPC}\; x \\ r_0' & = & \lambda x{:}\textbf{string}.\, \textbf{bind}\; y \;=\; r_0 \;\textbf{in}\; \textbf{return}@[\mathsf{K}]y\; x \end{array}$$

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

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

and singleton rule set

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

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

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

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

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

$$\begin{aligned} p_2' = \ & \textbf{bind}\ z \ = \ r_1 \\ & \textbf{in return}@[\mathsf{K}](z \ \texttt{"ab"}\ \mathsf{B}\ \textbf{sign}(\mathsf{B}, \mathsf{ReqRPC}\ \texttt{"ab"}). \end{aligned}$$

This term contains only B's signature, and hence B may be considered accountable for the action. Following Section 2.2's approach, B signed a request in every history consistent with this log entry.

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

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

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

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

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

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

$$\frac{\vdash t_2 \to t_2'}{\vdash \textbf{bind}\ x\ =\ t_1\ \textbf{in}\ t_2 \to \textbf{bind}\ x\ =\ t_1\ \textbf{in}\ t_2'}\ \textsc{R-bind2} \qquad \frac{\vdash t_1 \to t_1'}{\vdash t_1\ t_2 \to t_1'\ t_2}\ \textsc{R-app1}$$

$$\frac{\vdash t_2 \to t_2'}{\vdash t_1\ t_2 \to t_1\ t_2'}\ \textsc{R-app2} \qquad \frac{\vdash t_1 \to t_1'}{\vdash \langle t_1, t_2\rangle \to \langle t_1', t_2\rangle}\ \textsc{R-pair1} \qquad \frac{\vdash t_2 \to t_2'}{\vdash \langle t_1, t_2\rangle \to \langle t_1, t_2'\rangle}\ \textsc{R-pair2}$$

Figure 2.8: Reduction relation

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

Formal language properties

Subject reduction As the preceding example illustrates, proof simplification is a useful tool for audit. Following the Curry-Howard isomorphism, proof simplification corresponds to λ-calculus reductions on proof terms. This section provides several proof sketches; detailed proofs are given in the Appendix.

Aura$_0$ proof reduction is defined in Figure 2.8. For **bind**, in addition to the standard congruence, beta reduction, and commute rules as found in monadic languages, we also include a special beta reduction rule R-BINDS. The R-BINDS rule eliminates bound proofs that are never mentioned in the **bind**'s body. Rule R-BINDS permits simplification of terms like **bind** $x = \textbf{sign}(\mathsf{A}, P)$ **in** t, which are not subject to R-BINDT reductions. Aura$_0$ disallows reduction under **sign**, as signatures are intended to represent fixed objects realized, for example, via cryptographic means.

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

Lemma 2 (Subject Reduction). *If* $\vdash t \rightarrow t'$ *and* $\Sigma; \Gamma \vdash t : s$ *then* $\Sigma; \Gamma \vdash t' : s$.

Proof Sketch. By structural induction on the reduction relation. The R-BINDS case requires a strengthening lemma. This states that if variable x is bound in Γ but does not otherwise appear in the typing judgment, x may be removed and typing will be preserved. □

Proof normalization An expression is in *normal form* when it has no applicable reduction rules; as observed in Section 2.3, reducing a proof to its normal form can be quite useful for auditing. Proof normalization is most useful when the normalization process always terminates and every term has a unique normal form.

An expression t is *strongly normalizing* if application of any sequence of reduction rules to t always terminates. A language is strongly normalizing if all the terms in the language are strongly normalizing. We have proved that Aura$_0$ is strongly normalizing, which implies that any proof reduction strategy will terminate.

Lemma 3 (Strong Normalization). *Aura$_0$ is strongly normalizing.*

Proof Sketch. We prove that Aura$_0$ is strongly normalizing by translating Aura$_0$ to the Calculus of Constructions extended with dependent pairs, which is known to be strongly normalizing (Geuvers, 1995), in a way that preserves both types and reduction steps. The interesting cases are the translations of terms relating to the **says** monad: **return** expressions are dropped, **bind** expressions are translated to to lambda application, and a term **sign**(t_1, t_2) is translated to a variable whose type is the translation of t_2. To simulate reduction R-BINDC we extend CoC with the following new reduction rule and show the extended system is itself strongly normalizing.

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

Care must be taken to track dependencies in the types of variables introduced by translation. □

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

Lemma 4 (Confluence)**.** *If $t \rightarrow^* t_1$, and $t \rightarrow^* t_2$, then there exists t_3 such that $t_1 \rightarrow^* t_3$ and $t_2 \rightarrow^* t_3$.*

Proof Sketch. We first prove that Aura$_0$ is weakly confluent—if t reduces to t_1 and t_2 in one step, then both t_1 and t_2 reduce to some t_3 in zero or more steps. Weak confluences follows immediately from inspection of the reduction rules. We then apply Newman's lemma (Huet, 1980), which states strong normalization and weak confluence imply confluence. □

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

2.4 Examples

This section provides two examples of access control and audit in Aura$_0$.

Private medical records

Consider a database for storing medical records. For legal and ethical reasons, it is desirable to mediate access to patient data. In the United States the Health Insurance Portability and Accountability

Act (HIPAA) provides basic rules governing when and to whom a patient's medical records may be disclosed. On a consumer-oriented website, the department of Health and Human Services outlines HIPAA's privacy rules as follows.

> Only you or your personal representative has the right to access your records. A health care provider or health plan may send copies of your records to another provider or health plan as needed for treatment or payment or as authorized by you.[3]

This states two general principles. First, patients have the right delegate access to their records. Second, health care providers may access their patient's records. (Of course the actual policy is more complicated.) We can encode the simplified HIPAA policy as using two Aura$_0$ rules.

A patient chooses who may read his chart.

```
(patient: prin) → (a: prin) → (c: chart patient)
  → patient says (MayRead a c)
  → HIPAA says (MayRead a c)
```

Doctors can read their patients' charts.

```
(patient: prin) → (d: prin) → (DoctorOf patient d)
  → (c: chart patient)
  → HIPAA says (MayRead d c)
```

However this policy may actually be too strict! What happens when a patient stops breathing, her doctor is out of town, and the physician on call is not technically authorized—lacks proofs needed—to read her chart?

Weakening the privacy policy with the following rule addresses this scenario in a compelling way.

```
emergency: (patient: prin) → (a: prin)
                → (c: chart patient)
                → (reason: string)
                → HIPAA says (MayRead a c)
```

[3] Retrieved from `http://www.hhs.gov/ocr/privacy/hipaa/understanding/consumers/medicalrecords.html`, on October 15, 2009.

The emergency rule states that any principal may access a chart if they provide an explanation. In particular, the doctor on call can read the hypothetical patient's chart. Aura$_0$ proofs are logged, and this action can be reviewed later by social, administrative, or legal means.

Because Aura$_0$ provides a principled logging methodology, it reasonable to define escape hatches in access control policies and review their use after-the-fact. We claim this is a compelling design whenever the cost of a false deny exceeds the cost of a false access grant.

File system example

As a more detailed example, we consider a file system in which file access is authorized using Aura$_0$, and log entries consist of authorization proofs. In a traditional file, system access control decisions are made when a file is opened, and this examples primarily considers the open operation. Our open is intended to provide flexible access control on top of a system featuring a corresponding raw-open and associated constants:

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

$$\begin{array}{rr} \mathsf{RDONLY} : \mathsf{Mode} & \mathsf{WRONLY} : \mathsf{Mode} \\ \mathsf{APPEND} : \mathsf{Mode} & \mathsf{RDWR} : \mathsf{Mode} \end{array}$$

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

We can imagine that raw-open is part of the interface to an underlying file system with no notion of per-user access control or Aura$_0$ principals; it, of course, should not be exposed outside of the kernel. Taking inspiration from Unix, we define RDONLY, WRONLY, APPEND, and RDWR (the inhabitants of Mode), which specify whether to open a file for reading only, overwrite only, append only, or unrestricted reading and writing. Type FileDes is left abstract; it classifies file descriptors—unforgeable capabilities used to access the contents of opened files.

Figure 2.9 shows the interface to open, the extended signature of available predicates, and the rules used to construct the proofs of type K **says** OkToOpen $\langle m, f\rangle$ (for some file f and mode m) that open requires. OkToOpen and DidOpen are as specified in Section 2.2, and the other predicates have the obvious readings: Owns $A\ f$ states that the principal A owns the file f, ReqOpen $m\ f$ is a request to open file f with mode m, and Allow $A\ m\ s$ states that A should be allowed to open f

$\boxed{\text{Kernel Signature } \Sigma_K}$

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

$\boxed{\text{Kernel Interface } I_K}$

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

$\boxed{\text{Additional Types in Extended Signature } \Sigma_{ext}}$

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

$\boxed{\text{Rule Set } R\text{:}}$

$$\begin{aligned}
\mathsf{owner}f_1 &: \mathsf{K}\ \textbf{says}\ \mathsf{Owns\ A}\ f_1 \\
\mathsf{owner}f_2 &: \mathsf{K}\ \textbf{says}\ \mathsf{Owns\ B}\ f_2 \\
\mathsf{owner}f_3 &: \mathsf{K}\ \textbf{says}\ \mathsf{Owns\ C}\ f_3 \ldots
\end{aligned}$$

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

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

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

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

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

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

Figure 2.9: Types for the file system example

with mode m. (As we are not modeling authentication we will take it as given that all proofs of type A **says** ReqOpen $m\ f$ come from A; we discuss ways of enforcing this in Section 2.5.)

For each file f, the kernel provides a distinguished proof that some principal owns f. Notation owner$f$ abbreviates this term.

Rule delegate states that the kernel allows anyone to access a file with a particular mode if the owner of the file allows it. Rule owned relieves the file owner A from the need to create signatures of type A **says** Allow $A\ m\ f$ for files A owns, while readwrite allows a user who has acquired read and write permission for a file from different sources to open the file for reading and writing simultaneously. The rules read, write, and append do the reverse, allowing a user to drop from RDWR mode to RDONLY, WRONLY, or APPEND. These last four rules simply reflect semantic facts about constants of type Mode. Of the rules discussed in this paragraph only owned is derivable (from delegate); the rest must be defined using signatures.

With the rules given in Figure 2.9 and the other constructs of our logic it is also easy to create complex chains of delegation for file access. For example, Alice (A) may delegate full authority over any files she can access to Bob (B) with a signature of type

$$\mathsf{A}\ \textbf{says}\ ((C : \textbf{prin}) \to (m : \mathsf{Mode}) \to (f : \textbf{string}) \to$$
$$\mathsf{B}\ \textbf{says}\ \mathsf{Allow}\ C\ m\ f \to \mathsf{A}\ \textbf{says}\ \mathsf{Allow}\ C\ m\ f),$$

or she may restrict what Bob may do by adding further requirements on C, m, or f. She might restrict the delegation to files that she owns, or replace C with B to prevent Bob from granting access to anyone but himself. She could do both with a signature of type

$$\mathsf{A}\ \textbf{says}\ ((m : \mathsf{Mode}) \to (f : \textbf{string}) \to \mathsf{K}\ \textbf{says}\ \mathsf{Owns}\ \mathsf{A}\ f \to$$
$$\mathsf{B}\ \textbf{says}\ \mathsf{Allow}\ \mathsf{B}\ m\ f \to \mathsf{A}\ \textbf{says}\ \mathsf{Allow}\ \mathsf{B}\ m\ f).$$

As described in Section 2.2, the kernel logs the arguments to our interface functions whenever they are called. So far we have only one such function, open, and logging its arguments means keeping a record every time the system permits a file to be opened. Given the sort of delegation chains that the rules allow, it should be clear that the reason why an open operation is permitted can be rather complex, which provides a strong motivation for the logging of proofs.

One can easily imagine using logged proof terms—and in particular proof terms in normal form, as described in Section 2.3—to assist in assigning the blame for an unusual file access to the correct principals. For example, a single principal who carelessly delegates RDWR authority might be blamed more severely than two unrelated principals who unwittingly delegate RDONLY and WRONLY authority to someone who later makes use of readwrite. Examining the structure of proofs can once again allow an auditor to, in the terminology of Section 2.2, rule out certain histories.

We can also see how logging proofs might allow a system administrator to debug the rule set. The rules in Figure 2.9 might well be supplemented with, for example

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

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

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

Rule maybe, on the other hand, is a more subtle—it makes the ability to overwrite a file strictly more powerful than the ability to append, even in the absence of the ability to read. This is useful and allows, for example, efficient completion of prematurely aborted write operations. However, maybe might defy user expectations and potentially allow violations of institutional policy. In the event of such a violation, the administrator can examine logs to determine how and if maybe rules were abused and take appropriate action.

The following shows that Aura_0 policies can benefit operations other than open.

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

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

Ownership File creation and deletion are certainly operations that should require authorization, and they are especially interesting due to their interaction with the Owns predicate. The creation of file f by principal A should introduce a rule owner$f$: Owns A f into the rule set, while the deletion of a file should remove said rule; a means of transferring file ownership would also be desirable. This can amount to treating a subset of our rules as a protected resource in its own right, with a protected interface to these rules and further rules concerning the circumstances under which access to this new resource should be granted. An alternate approach is to dispense with ownership rules completely and instead use signed objects and signature revocation to represent ownership. This is discussed further in Section 2.5,

2.5 Discussion

Signature implementation Thus far we have treated signatures as abstract objects that may only be created by principals or programs with sufficient authority. There are at least two appealing implementation strategies for signatures.

The first approach is cryptographic: a **sign** object can be represented by a digital signature from a public-key cryptosystem. Each principal must be associated with a well known public key and in possession of its private counterpart; implementing rule T-SIGN reduces to calling a digital signature verification function. The cryptographic scheme is well suited for distributed systems with mutual distrust.

A decision remains: the signature $\textbf{sign}(\mathsf{A}, P)$ can be represented either as a tuple containing the cryptographic signature along with A and P in plaintext, or as the cryptographic signature alone. In the latter case signatures are small (potentially constructed from a hash of the contents), but recovering the text of a proposition from its proof (i.e., doing type inference) may not be possible. In the former case, inference is trivial, but proofs are generally large. Note that proof checking of **sign**s in either case involves validating digital signatures, a polynomial time operation.

The prototype Aura interpreter described in Chapter 3 follows the former approach: signature objects do not contain plaintext representations of their propositions. Additionally, as an optimization, actual digital signatures are generated on demand before being serialized for output. (The benefits of discarding plaintext are not realized until after signature generation; this scheme is intended to reduce log size and processor use but not heap size.)

An alternative implementation of signatures assumes that all principals trust some *moderator*, who maintains a table of signatures as abstract data values; each **sign** may then be represented as an index into the moderator's table. Such indices can be small while still allowing for easy type inference, but such a scheme requires a closed system with a mutually trusted component. In a small system, the moderator can be the kernel itself, but a larger system might contain several kernels protecting different resources and administered by disparate organizations, in which case finding a suitable moderator may be quite difficult.

Proof normalization Proofs in normal form are useful for audit because they provide a particularly clear view of authorization decisions. Normalization, however, is an expensive operation—

even for simply typed lambda calculus, the worst-case lower-bound on the complexity of the normalization is on the order of $\mathit{exp}(2, n)$, where $\mathit{exp}(2, 0) = 1$, $\mathit{exp}(2, n) = 2^{\mathit{exp}(2,(n-1))}$, etc., and n is the size of the term (Schwichtenberg, 1989). Furthermore, the size of a normalized proof can grow to $\mathit{exp}(2, n)$ as well. In practice, well-behaved proof producers will likely create proofs that are simple to normalize. While normalization is expensive, checking that a proof is normal form requires only linear time in the proof size. Consequently, where, and if, to normalize proofs is an engineering question whose solution may be application dependent.

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

Authentication In Section 3.5 we assumed that signatures of type A **says** ReqOpen $m\ f$ are always sent from A. Such an assumption is necessary because we are not currently modeling any form of authentication—or even the association of a principal with a running program—but a more realistic solution is needed when moving beyond the scope of this work. For example, communication between programs running on behalf of different principals could take place over channel endpoints with types that depend on the principal on the other end of the channel.

Of course, when this communication is between different machines on an inherently insecure network, problems of secure authentication become non-trivial, as we must implement a secure channel on top of an insecure one. In practice this is done with cryptography, and one of the long-term goals of the Aura project is to elegantly integrate cryptographic methods with the type system, following the work of, for example, Fournet et al. (2007).

2.6 Related Work

This section discusses previous research on audit logging and proof-based access control.

Audit logging

What information should programs log? How can logs be interpreted for Audit? Who can be held accountable when programs fail? We are aware of surprisingly little literature that examines these questions.

Wee (1995) implemented the Logging and Auditing File System (LAFS), a practical system which shares several architectural elements with Aura$_0$. LAFS uses a lightweight daemon, analogous to our kernel, to wrap NFS file systems; like our kernel, the LAFS daemon forwards all requests to the underlying resources. Both systems also configure policy using sets of rules defined outside the trusted computing base. The systems differ in three key respects. First, the LAFS policy language is too weak to express many Aura$_0$ policies. Second, Aura$_0$ requires some privileged K **says** rules to bootstrap a policy, while LAFS can be completely configured with non-privileged policies. Third, the LAFS interface is designed to be transparent to application code and does not provide any access control properties; instead LAFS logs—but does not prevent—rule violations.

Cederquist and colleagues (2005) describe a distributed system architecture with discretionary logging and no reference monitor. In this system agents—i.e. principals—may choose to enter proofs (written in a first-order natural deduction style logic) into a trusted log when performing actions. Cederquist *et al.* formalize accountability such that agents are guilty until proved innocent—that is, agents use log entries to reduce the quantity of actions for which they can be held accountable. This relies on the ability of some authority to independently observe certain actions; such observations are necessary to begin the audit process.

Etalle and Winsborough (2007) present a role-based, access-control logic intended primarily for answering questions about accountability. Like us, they observe that allowing deviations from access-control policy, coupled with a audit mechanism, enables greater flexibility than simply enforcing access-control rules.

Authorization logics and proof-carrying access control

Earlier work on proof-carrying access control (Abadi et al., 1993; Appel and Felten, 1999; Cederquist et al., 2005; Bauer et al., 2005a,b; Fournet et al., 2005; Abadi, 2007; DeYoung et al., 2008; Guts et al., 2009) recognized the importance of **says** and provided a variety of interpretations for it. Garg and Pfenning (2006) and, later, Abadi (2006) introduced the treatment of **says** as an indexed monad. Both systems (Garg and Pfenning, 2006; Abadi et al., 1999) also enjoy a crucial noninterference property: in the absence of delegation, nothing B says can cause A to say false.

Aura adopts ideas from this prior work, especially Abadi's DCC, for several reasons. First, DCC proofs are lambda-terms, a fact we exploit to closely couple the **Prop** and **Type** universes. Second, DCC is strong enough to define important authorization concepts, such as the ActsFor relation and the hand-off rule (a **says** b ActsFor a) → b ActsFor a. Third, DCC-style noninterference provides a crisp technical framework for thinking about authorization in the presence of mutually distrusting principals.

Aura also embodies substantial changes to the DCC model. The addition of dependent types enhances the expressiveness of DCC, and the addition of **sign** allows for a robust distributed interpretation of **says**. Aura's treatment of principals as terms, as opposed to members of a special index set, enables quantification over principals. Lastly, Aura eliminates DCC's built-in acts-for lattice, which can be encoded as described in Section 2.3, along with the protects relation, which allows additional commutation and simplification of **says** with regard to that lattice. Dropping the protects relation eliminates unintuitive DCC tautologies, such as $(\mathsf{A}\ \textbf{says}\ \mathsf{B}\ \textbf{says}\ P) \to (\mathsf{B}\ \textbf{says}\ \mathsf{A}\ \textbf{says}\ P)$, and ensures desired **says** manipulations are explicitly recorded in proofs. Abadi (2007) defined a similar DCC variant, dubbing it cut-down DCC, or CDD.

DeYoung, Garg, and Pfenning (2008) 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.

Our work is closely related to Fournet, Gordon, and Maffeis's (2005; 2007) research on authorization in distributed systems They work with a π-calculus-based model of computation, which, like Aura, uses dependent types to express access control properties. Fournet and colleagues focus on security properties that are maintained during execution, which are reflected into the type system using static theorem proving and a type constructor **Ok**. The inhabitants of **Ok**, however, do not contain dynamic information and cannot be logged for later audit. Additionally, while Aura treats signing abstractly, Fournet and colleagues' type system (and computation model) can explicitly discuss cryptographic operations. 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.

In a related project, Guts, Fournet, and Zappa Nardelli (2009) explore an audit mechanism for the F7 extension to F♯. F7 uses **assume** and **assert** expressions that statically describe a program's requirements (assertions) and steps taken to fulfill those requirements (assumptions). These expressions very loosely correspond to program points that introduce and consume **Ok** types. Guts and colleagues add an **audit** construct that can replace **assume** to indicate the program must record evidence proving the assumed property holds. Unlike in Aura, usage of **audit** is discretionary. This may reduce logging of uninteresting proofs, but may require additional programmer vigilance to ensure comprehensive logging.

Proof carrying access control has been field tested in Grey project (Bauer et al., 2005a,b). In this system, smart phones build proofs which can be used to open office doors or log into computers. The Grey architecture shares structural similarities with the Aura model: in Grey, proof-generating devices, like our applications, need not reside in the trusted computing base. And both systems use expressive foundational logics to define policies—higher-order logic in Grey's case (Church, 1940). To make proof search effective, Bauer at colleagues suggest using cut-down fragments of higher order logic for expressing particular rule sets and using a distributed, tactic-based proof search algorithm. While later chapters of this document will treat Aura as a tightly coupled authorization logic and computation language, Grey's logic is not integrated with a computation language.

Chapter 3

Aura: A Language for Authorization and Audit

This chapter describes the Aura programming language's design and key metatheoretic properties. The full language includes many interesting features not found in Chapter 2's Aura$_0$, particularly inductive types and a complete computational sub-language. Aura's metatheory—including syntactic soundness and decidability of typing checking—is formalized in Coq. Additionally, the chapter describes a prototype interpreter for the language and a sample Aura program.

This chapter reports on joint work with Limin Jia, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic.

3.1 Introduction

This chapter presents the design of Aura, a domain-specific programming language that incorporates a constructive authorization logic based on Aura$_0$ as part of its type system. Rather than mediating between computations and policy statements written in two largely distinct languages, Aura provides a uniform treatment of these concepts. For example, consider the following signature that declares a simple API for playing songs.

jazzPolicy : **self says** (s : Song) → (p: **prin**) → isJazz s → MayPlay p s
playFor: (s : Song) → (p : **prin**) → **pf** (**self says** MayPlay p s) → Unit.

As hinted by the types, Aura can express both access control policy statements and procedures guarded by such policies. Note also that a single → constructor is used to represent both logical implication and function arrow. In contrast Aura$_0$ makes a distinction between → and ⇒. This is symbolic of a design philosophy used in Aura that seeks to provide uniformity across apparently diverse language constructs.

Aura features a full computation model including higher-order functions, polymorphism, and recursive algebraic datatypes. This model also makes explicit how security-relevant implementation details—like the creation of audit trails or the cryptographic interpretation of certain logical statements—can be handled automatically with little programmer intervention. It is not enough to to naively mix authorization logic and functional programming features. Instead, Aura's type system is specialized to enable dependent types in the presence of non-terminating computations, and the language includes specialized constructs for signature generation and dynamic type refinement.

This chapter's main contributions are as follows:

- It presents the design of Aura, a language unifying first-class, dependent authorization policies with functional programming.
- It describes machine-checked proofs that Aura's type system is sound and decidable.
- It describes a prototype typechecker and interpreter for Aura and gives sample programs.

Typical dependently typed languages (see Section 3.6) use types to encode precise program specifications. Constructing associated proof objects is computationally hard and typically requires the help of a theorem prover or interactive proof assistant. 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 static program verification or general-purpose theorem proving.

Aura's metatheory is verified in the Coq proof assistant and encompasses the complete language, 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 chapter focuses on the novel core features of Aura. The next section introduces Aura's programming model and illustrates its novel features by example. Section 3.3 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. Finally, Section 3.4 describes a prototype implementation of an interpreter and typechecker for Aura.

3.2 Programming in Aura

Aura is intended to be used to implement reference monitors (Bishop, 2002) 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 after-the-fact audit. Chapter 2 covered audit in detail, and logging will only be discussed briefly here. This chapter is focused 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.

This chapter to Chapter 1's jukebox-server example. The full server code, given in Section 3.5, and the rest of this section will use relevant functions and propositions to motivate and discuss parts of Aura's design.

Authority and authorization logic

This section examines Aura's handling of principals and policy assertions. Principals in Aura have type **prin** and represent distinct components of a software system. They may correspond to human users or system components such as an operating system kernel or a particular server. Formally, principals are treated as special values in Aura; they are characterized by their ability to index the family of **says** monads.

Aura's intended semantics associates each principal with a unique private key used to cryptographically sign messages. In such a setting objects of the form **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. At runtime, an Aura program has access to a single private key. If the program has access to A's key, we say that program is running with A's authority.

Aura 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 extended to support dynamically generated principals, type-based analysis might suffice to keep private keys private.

As in Aura$_0$, **says**-propositions may be inhabited by **return**, **bind**, and **sign** terms. Additionally, Aura allows for the creation of fresh signatures using the new construct **say**. Intuitively, when run with authority A, expression **say** P reduces to **sign**(A, P). When signatures are represented cryptographically, this requires access to A's private key.

Aura's formal definition handles runtime keys and authority implicitly. Instead, the language semantics are given from a local perspective, and a distinguished value, **self**, represents whichever principal actually runs the program. This local viewpoint manifests itself in the type of **say**,

$$\textbf{say}\ \mathsf{P} : \ \textbf{self}\ \ \textbf{says}\ \mathsf{P}.$$

Chapter 4's formalism provides a more explicit account of runtime authority. There, both the reduction relation and the **say** operator are annotated with authorities, and a type-and-effect–style system ensures that these annotations are properly aligned. While this adds complexity, it facilitates reasoning about systems from a global perspective.

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 (Curry et al., 1958; Howard, 1980) 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.

Aura incorporates dependent types directly—in contrast to, for example, using GADTs (Peyton Jones et al., 2006) or static equality proofs (Sulzmann et al., 2007) 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.

Inspired by the Calculus of Inductive Constructions (Coquand and Huet, 1988), Aura has separate universes, **Type** and **Prop**, that classify computational objects and proofs. (Aura's **Type** universe corresponds to CIC's **Set**.) The introduction's MayPlay proposition constructor, for instance, has type **prin** $\rightarrow$ Song $\rightarrow$ **Prop**. Both types of kind **Type** and propositions of kind **Prop** describe data that may be available at runtime. Propositions, however, are computation-free: they never reduce and Aura does not employ type-level reduction during typechecking. Consequently, type-level terms may only depend on (or mention) values (well-typed normal forms) not arbitrary computations.

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

if **self** = a **then** e_1 **else** e_2,

the fact that **self** equals a is automatically made available while typechecking e_1, and proofs of type **self** **says** P can be cast to type a **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 computation dependent on a program's runtime authority. On the other hand, **sign**(a, P) should be of type a **says** P, which, like P, is of kind **Prop**. To type both **says** and **say**, we introduce the modality **pf** : **Prop** $\rightarrow$ **Type** and give **say** P the type **pf** (**self** **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 the cryptographic interpretation of the **says** monad, namely that one often thinks of digitally signing data, whereas **sign** only works with propositions. Dependent types resolve this issue, using assertions that mention data. For example, while **sign**(a, 42) is not well formed,

```
pair 312 (sign(a, Good 42))
```

where

```
pair: (x: int) → A says (Good x) → Pair
```

is well formed and associates a data value with a signature about the value. Only signing propositions is compelling, because a digital signature on raw data lacks intrinsic meaning but signed propositions associate data with metadata.

Auditing in Aura

Passing proofs at runtime is also useful for after-the-fact auditing of Aura programs. The full details are given in Chapter 2, but recall 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

```
λs: Song. λp: prin.
  λproof: pf (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.

3.3 The Aura Core Language

This section presents the chapter's main technical contributions, 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 (Lee et al., 2007): 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.

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 (Barendregt, 1992) 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 (Peyton Jones and Meijer, 1997), which also adopts this compact representation.

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

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

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 **match** t_1 t_2 **with** $\{b\}$, term t_1 is 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 $\langle t_1 : t_2 \rangle$ lets t_1 be considered to have type t_2; typechecking ensures this cannot fail.

To express and reason about access control, Aura extends the core syntax above with additional terms. 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 by the following grammar. (Appendix C, about Chapter 4's language, shows how to represent such constructs in Coq.)

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

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

Figures 3.1 and 3.2 show the term typechecking rules. For now, we elide the rules for typechecking signatures and branches and briefly 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.

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

Binding $x \sim (v_1 = v_2){:}t$ indicates that values v_1 and v_2 have type t and at run-time v_1 and v_2 will be equal. (The variable x is unimportant, but naming the equality assumption allows for a uniform environment representation.)

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 B; here we explain signatures via examples.

Data definitions may be parametrized, and the familiar polymorphic list declaration is written:

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

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

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

$$\frac{S; E, x\!:\!t_1 \vdash t_2 : k_2 \qquad k_2 \in \{\textsf{\textbf{Type}}, \textsf{\textbf{Prop}}, \textsf{\textbf{Kind}}\}}{S; E \vdash (x\!:\!t_1) \to t_2 : k_2}\ \textsc{Wf-tm-arr}$$

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

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

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

Figure 3.1: Aura typing rules for standard functional language constructs.

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:

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

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

$$\frac{S; E \vdash a : \textbf{\textsf{prin}} \qquad S; E \vdash P : \textbf{\textsf{Prop}}}{S; E \vdash a\ \textbf{\textsf{says}}\ P : \textbf{\textsf{Prop}}}\ \textsc{Wf-tm-says}$$

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

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

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

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

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

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

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

Figure 3.2: Aura typing rules for access control constructs.

Aura's type system conservatively constrains **Prop** definitions to be inductive by disallowing negative occurrences of **Prop** constructors. While this is restrictive, maintaining some positivity constraint is essential to the consistency of the logic. It would otherwise 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 3.2, are special constants that construct **Prop**s:

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, and principals may sign assertions without compromising their **says** monad's consistency. In this way, assertions are similar to abstract types or type variables.

Core term typing

Figures 3.1 and 3.2 present the main typing rules for Aura.

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 word "type" for any classifier—**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, $(x : \textsf{\textbf{Type}}) \rightarrow$ **Type** is legal whereas $(x : \textsf{\textbf{Type}}) \rightarrow$ **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 typeable 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 (i.e., for terms classified by **Kind**). Such reductions, while useful for verification, are superfluous in this setting. Leaving the reductions out simplifies typechecking and Aura's metatheory.

The interesting part of the WF-TM-APP rule is the side condition requiring either that t_2 is a value ($val(t_2)$) or that u does not depend on x—that is, $x \notin fv(u)$. This restriction has the effect that, while Aura seems to be quite liberal with respect to the dependencies allowed by well-formed $(x : s) \rightarrow t$ 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 expression.

These restrictions mean 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. See Appendix B for the formal definition of the value relation.

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 parametrized 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.

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 3.2, Aura uses the principal-indexed **says** monad to express access control policies. The proposition a **says** P means that principal a has asserted proposition P (either directly or indirectly). The expression $\textbf{return}_s\ a\ p$ introduces proofs into the a **says** monad, and $\textbf{bind}_s\ e_1\ e_2$ 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 (Abadi, 2007). Aura, is mostly closely related to Abadi's "cut-down" DCC variant, as Aura drops DCC's label lattice in favor of explicit delegation among principals.

The expression $\textbf{sign}(a, P)$ witnesses the assertion of proposition P by principal a (WF-TM-SIGN). Since $\textbf{sign}(a, P)$ 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. Additionally the declaration signature S must be available where the **sign** object is interpreted. When two Aura programs exchange proofs they must agree on a declaration signature. (Strictly speaking, it may be sound for the signatures to differ in ways that do not affect the meaning of the **sign** object (Rossberg, 2007; Sewell et al., 2007; Nanevski et al., 2008).)

Creating $\textbf{sign}(a, P)$ requires a's authority. Aura models the authority of a running program with the principal constant **self**. The **say** P operation creates an object of type **pf self says** P. This operation creates the signed assertion $\textbf{sign}(\textbf{self}, P)$ and injects it as a proof term for further manipulation (see WF-TM-SAY). Intuitively **self** represents the current private key. Chapter 4 provides a more explicit account of runtime authority.

Aura uses the **pf** monad to wrap access control proofs as program values. Term $\textbf{return}_p\ p$ turns proof $p : P$ into an expression of type **pf** P. The $\textbf{bind}_p$ construct allows a computation to compose proofs (rules WF-TM-PF-RET and WF-TM-PF-BIND). Maintaining a separation between proofs and computations is necessary to prevent effectful program expressions from being confused with genuine proof terms. For intuition consider **say** P. This term's evaluation depends on the current runtime authority—that is **say** has a runtime effect. It would be incorrect to put **say** in the **Prop** universe; therefore, **say** P is classified by **pf** (**self says** P), not just **self says** P.

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

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

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

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

$$\frac{\textit{converts}\ E\ s_1\ t_1 \qquad \textit{converts}\ E\ s_2\ t_2}{\textit{converts}\ E\ ((x\,{:}\,s_1) \rightarrow s_2)\ ((x\,{:}\,t_1) \rightarrow t_2)}\ \textsc{Conv-arr}$$

Figure 3.3: Conversion

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 for full-spectrum dependently typed languages.

Aura addresses this with a conditional construct. Dynamically, **if** $v_1 = v_2$ **then** e_1 **else** e_2 steps to e_1 when v_1 and v_2 are equal, and the expression yields e_2 otherwise. Statically, rule WF-TM-IF typechecks the **then** branch 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 (Coq, 2006), Agda (Norell, 2007), and Epigram (McBride, 2005).

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-parametrized **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:

```
data Song: Type {
  | freebird : Song
  | ironman: 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 arbitrary datatypes not containing functions.

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 (Barendregt, 1992), 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 3.3, is of course reflexive, symmetric, and transitive; the key rule is CONV-AXIOM, which uses equality assumptions in the environment. For instance, under assumption $x = \textbf{self}$, term $x\,\textbf{says}\,P$ converts to $\textbf{self}\,\textbf{says}\,P$. 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 embedded 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.

Evaluation rules

Figure 3.4 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 $\textbf{say}\,P$ creates a proof that principal **self** has asserted that proposition P is true; it evaluates to an assertion signed by principal **self**. There are two possibilities in the evaluation of $\textbf{if}\ v_1 = v_2\ \textbf{then}\ e_1\ \textbf{else}\ e_2$: 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

$\boxed{t \mapsto t'}$

$$\frac{\mathit{val}(v)}{(\lambda x\!:\!t.\ e)\, v \mapsto \{v/x\}e}\ \textsc{App} \qquad \frac{}{\textsf{\textbf{bind}}_p\ (\textsf{\textbf{return}}_p\ e_1)\ e_2 \mapsto e_2\ e_1}\ \textsc{Pf-bind}$$

$$\frac{}{\textsf{\textbf{say}}\, P \mapsto \textsf{\textbf{return}}_p\ (\textsf{\textbf{sign}}(\textsf{\textbf{self}}, P))}\ \textsc{Say} \qquad \frac{v_1 = v_2}{\textsf{\textbf{if}}\ v_1 = v_2\ \textsf{\textbf{then}}\ e_1\ \textsf{\textbf{else}}\ e_2 \mapsto e_1}\ \textsc{If-eq}$$

$$\frac{v_1 \neq v_2}{\textsf{\textbf{if}}\ v_1 = v_2\ \textsf{\textbf{then}}\ e_1\ \textsf{\textbf{else}}\ e_2 \mapsto e_2}\ \textsc{If-neq} \qquad \frac{\mathit{val}(v)}{\langle v : t\rangle \mapsto v}\ \textsc{Cast}$$

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

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

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

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

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

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

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

Figure 3.4: Reduction Relation

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 $(\mathsf{cons},\ 1)$.

Metatheory

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

Theorem 5 (Preservation)**.** *If $S; \cdot \vdash e : t$ and $e \mapsto e'$, then $S; \cdot \vdash e' : t$.*

Theorem 6 (Progress)**.** *If $S; \cdot \vdash e : t$ then either $val(e)$ or exists e' such that $e \mapsto e'$.*

We have also proved that typechecking in Aura is decidable.

Theorem 7 (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, to prove the preservation of pattern matching, we examine 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. Appendix B contains the fully explicit presentation of Aura's type system.

Finally, Jia and Zdancewic (2009) showed independently that Aura is strongly normalizing.

3.4 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 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, encoding syntax and semantics with a variant of the locally nameless representation (Aydemir et al., 2008). 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 Coq proofs is relatively rudimentary; we did not devote much time to writing 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. As later discovered when mechanizing Aura$_{\text{conf}}$, additional tactic support reduces the cost of design iteration and further magnifies the benefits of mechanization.

Because Aura is a superset of system F with inductively defined datatypes, we hope that the Aura proof scripts will be a useful starting point for future mechanized soundness proofs.

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.

3.5 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 Figure 3.5. The example program typechecks in the prototype Aura interpreter and uses some of the language extensions discussed in Section 3.4. 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 11 and 12. 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 52) 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 98 and 117). These rules use monadic bind and return operations to change the placement of **pf** and **says** type constructors in 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 142.

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

(∗ Introduce an atomic datatype representing songs ∗)
data Song : Type {
  | warpigs: Song
  | heartbreaker : Song
 }


(∗ Base predicates to describe ownership and listening rights .∗)
assert Owns : prin → Song → Prop;
assert MayPlay : prin → Song → Prop;


(∗ Existential type for song ownership ∗)
data OwnerRecord : Type {
 | ownerRecord : (p: prin) → (s: Song) →
                   (pf (self says (Owns p s))) → OwnerRecord }

(∗ List of songs owners. This list will be created at runtime (using ”say”),
   but could also be read from a separate configuration file with
   appropriate signatures. ∗)
let ownerlist : ( List OwnerRecord) =
  (cons OwnerRecord
        (ownerRecord self heartbreaker (say (Owns self heartbreaker)))
     ( nil OwnerRecord))
in


(∗ Import foreign functions ∗)
prim static method println : String → Unit = System.Console.WriteLine in
prim static method playWav : String → Unit =
                                   Sol.InteropTestLib.Featureful.playWav in

let seq : (t:Type) → Unit → t → t = (λt:Type. λa:Unit. λb:t. b) in
let ignore : (t:Type) → t → Unit = (λt:Type. λv:t. unit) in

let printMessage : (s: String) → Unit =
  λs: String.
  seq Unit ( println ””) (seq Unit ( println ””) ( println s)) in

let findWavPath : (s: Song) → String =
  λs: Song .
  match s with (String) {
   | heartbreaker → ”media/heartbreaker.wav”
   | warpigs → ”media/warpigs.wav”
  }
in

```

Figure 3.5: Aura code for a music store

```
(∗ Rule stating that owners may delegate their right to play songs.
   This is inhabited using ”say.” ∗)
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

(∗ Use the foreign function interface to play a song ∗)
let playFor : (s: Song) → (p: prin) →
                          (pf (self says (MayPlay p s))) → Unit =
  λs: Song . λp: prin . λproof: (pf (self says (MayPlay p s))) .
 seq (Unit) (printMessage ”∗∗∗∗PLAYING SONG∗∗∗∗”)
      (playWav (findWavPath s))
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 .

    fun rec : (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 .
                (∗seq (Maybe (pf (self says Owns p s))) (println ”cons”)∗) (
                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
                  } ) }
        in rec ownerRecords end
in
```

Aura code for a music store (cont.)

```
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

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

```

Aura code for a music store (cont.)

```
134 let handleRequest: (s: Song) → (p: prin) → (o: prin) →
135                     ( List OwnerRecord) →
136                     (delPf: pf (o says (MayPlay p s))) → Unit =
137   λs: Song. λp: prin. λo: prin. λl: List OwnerRecord.
138     λdelPf: pf (o says (MayPlay p s)).
139     match (getOwnerProof s o l) with Unit {
140     | nothing → printMessage ”∗∗∗∗PROOF SEARCH FAILED. ACCESS DENIED.∗∗∗∗”
141     | just → λx: (pf (self says (Owns o s))).
142                       playFor s p (shareRule′′ o p s x delPf)
143     }
144 in
145
146 let selfMayPlayGen : (s: Song) → pf (self says (MayPlay self s)) =
147   λs: Song. say (MayPlay self s)
148 in
149
150 handleRequest heartbreaker self self ownerlist (selfMayPlayGen heartbreaker)
```

Aura code for a music store (cont.)

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 60. 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. playFor is implemented in terms of a foreign function imported (line 32) from .Net.

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 134) is straightforward. There are two interesting things to note. First, handleRequest takes a database of owner information expressed as a list of OwnerRecords. OwnerRecord (line 16) 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 139 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 73) performs this type refinement in several steps. It uses the fixpoint combinator (line 77) 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 87 and 93. 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 Songs 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 return it packaged as a Maybe. If either dynamic check fails we repeat again, and, if no match is found, we eventually return Nothing.

In this example the server itself plays a song itself (line 150). The example in Chapter 4 shows a server that communicates with clients in an interesting way.

3.6 Related Work

Aura's goals were strongly influenced by earlier programming languages with access-control features, and its design was inspired by dependent type systems.

Language-based access control

The trust management system PolicyMaker (Blaze et al., 1999b) 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 are similar in purpose, but not realization, to **says** in Aura. And, while validity of Aura propositions is tested by type checking, validity in PolicyMaker is tested by *evaluation*; this represents a fundamentally different approach to logic. Despite this, Aura and trust management systems share a common design principle: a trusted, application-independent component checks proofs, but proof search can offloaded to untrusted components. The ideas in PolicyMaker have been refined in KeyNote (Blaze et al., 1999b,a) and REFEREE (Chu et al., 1997).

The Fable language (Swamy et al., 2008) 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. 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) (Barendregt, 1992; Coquand and Huet, 1988). 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 (Coq, 2006), 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 and Hoare Type Theory (Nanevski et al., 2006), Agda (Norell, 2007), and Epigram (McBride, 2005) are intended to support general purpose program verification and usually require that the programmer construct proofs interactively. By contrast, Dependent ML (Xi and Pfenning, 1998), ATS (Xi and Pfenning, 1998; Xi, 2004), and RSP1 (Westbrook et al., 2005) 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 (Augustsson, 1998) 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 (2000) 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 (1997) describe the Henk typed intermediate language. 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.

Chapter 4

Confidentiality in Aura

This chapter introduces Aura$_{\text{conf}}$, a confidentially extension to Aura. The extension is based on the full language defined in Chapter 3. In the following we describe Aura$_{\text{conf}}$ informally, define the language, and discuss formalized proofs of key metatheoretic properties including noninterference.

4.1 Introduction

Thus far, this thesis has discussed Aura as a platform for programming with access control and audit. However these security properties are not sufficient for programs that deal with confidential data. For instance, applications in medical, financial, and legal arenas need to be able process and, as appropriate, share secrets.

This chapter introduces Aura$_{\text{conf}}$, an extension to Aura that provides programmers with a principled set of tools for handling secret information. The goals of this design are as follows.

- To establish a natural connection between confidential language expressions and cryptography.

- To leverage Aura's expressive core features to provide a cohesive and useful design.

- To provide technical mechanisms for associating decryption failures with proof objects that can be logged for later analysis.

Mixing an informative type system with encryption exposes a fundamental tension. Type systems gain power—the ability to prevent errors and uphold invariants—by exploiting precise information about a program's terms. In contrast, the point of encryption is to obscure information in certain contexts.

This tension has several technical manifestations. First, typing is relative; each principal has its own, local notion of what is well-typed. This is desirable because it accounts for the following real phenomenon. To Alice all arbitrary, unknown bit strings are plausibly encrypted messages for Bob—elements of the Aura$_{\text{conf}}$ type **int for** Bob. In contrast Bob can tell which bit strings are well-formed at that type, and which are garbage.

Second, typing exhibits a hysteretic, or path dependent, effect. When Alice creates a new ciphertext for Bob, she transforms a perfectly legible piece of abstract syntax into an opaque binary blob. In order for type preservation to hold during this process, Alice's computation must annotate the ciphertext and, as a side-effect, record information to validate the annotation in the future.

Third, resolving the above issues requires a precise treatment of public keys, both at compile time and at runtime. Discussing key availability at different hosts requires ideas from modal type theories (Jia and Walker, 2004; Murphy, 2008). Ensuring that needed keys are available dynamically requires type-and-effect analysis (Lucassen and Gifford, 1988; Talpin and Jouvelot, 1992; Washburn, 2005). (In principle other techniques could be used, but the concepts involved are essential.)

The Aura$_{\text{conf}}$ language resolves the tensions indicated above and helps programmers to safely handle confidential data.

4.2 Confidential Computations and the For-Monad

This section provides an informal introduction to Aura$_{\text{conf}}$'s new features.

In Aura$_{\text{conf}}$ secrets are protected with an indexed confidentiality monad. And a confidential integer intended only for Alice can be given type (**int for** Alice). As expected, values of this type are constructed using the following monadic return operator.

return Alice 42 : **int for** Alice

This expression evaluates by encrypting 42 with Alice's public key, yielding a blob of ciphertext, written $\mathcal{E}$(Alice, 42, 0x2b63), with an additional annotation that will be discussed shortly. The number 0x2b63 is a random value inserted by the encryption algorithm to ensure that encrypting identical plaintexts does not yield identical ciphertexts. Code running on any host should be able to perform this operation, as it uses only Alice's public key and needs no access to private keys. A program running with Alice's private key may decrypt and declassify the ciphertext as follows.

```
run (return Alice 42) : int
```

Additionally, when given a value of type **int for** Alice, Aura$_{\text{conf}}$ programs can use a bind operator to produce a new encrypted computation, also for Alice, based on the existing secret.

```
bind (int for Alice)
    (return Alice 42)
    (λ{Alice} x: int . return Alice (x ∗ 2))
  : int for Alice
```

When Alice runs the resulting encrypted computation, she will decrypt the 42 before supplying it to the decryption of the function. Ignore for now the {Alice} component of the λ; this is an effect annotation and will be defined and discussed later.

As illustrated above, for-monad operators treat their arguments lazily. Imagine for the moment that we could use homomorphic encryption (Rappe, 2004; Gentry, 2009) to allow for-bound computations to be applied eagerly. (I am not aware, by the way, of any practically efficient homomorphic encryption scheme.) An eager for-bind would permit curious adversaries to probe encrypted objects using functions that diverge on known inputs. Giving the for-monad a lazy semantics eliminates this termination channel.

While the dynamic semantics of encryption are straightforward, they pose a substantial problem for typechecking. Consider a machine running on Bob's behalf that performs the above encryption for Alice. A sound type system should satisfy subject reduction and be able to relate ciphertext $\mathcal{E}$(Alice, 42, 0x2b63) with type **int for** Alice. The entire point of encryption is to ensure that users other than Alice cannot meaningfully inspect the ciphertext, and Bob has no way to decompose and examine the newly created object.

Aura$_{\text{conf}}$ resolves this tension as follows. Ciphertexts may be annotated with one of two forms of typing metadata. First, the term

cast $\mathcal{E}$(Alice, 42, 0x2b63) **to** (**int for** Alice) : **int for** Alice

is a *true cast*—a form of type coercion allowed only when semantic evidence indicates that the cast is "correct." A true cast typechecks when the ciphertext is a known value with known provenance. Whenever Bob's program creates a ciphertext, it records a *fact* associating the new ciphertext with the appropriate type. As evaluation proceeds, programs accumulate a context of facts which are used to typecheck known ciphertexts. We assume fact contexts are part of a host's local state and are not shared between different principals. (Groups of colluding attackers can be modeled as a single principal that obeys this rule.) True casts are also permitted when the typechecker can *statically* access an appropriate decryption key. Thus the above cast can be typechecked on Bob's machine, where it originated, as well as on Alice's machine, where it will be used. Evaluating a **return** yields a ciphertext annotated with a true cast.

True casts alone are insufficient for writing some protocols. Consider two programs, running with Bob and Charlie's authority respectively, jointly constructing an **int for** Alice using the **return** and **bind** operators. In particular, Charlie's program may need to **bind** a ciphertext previously created by Bob. Because facts are not shared between different principals, and because Charlie cannot access Alice's private key, there's no way Charlie will be able to typecheck the ciphertext annotated with a true cast. Instead, Charlie's program will need to work with a justified cast,

cast c **to** (**int for** Alice) **blaming** p : **int for** Alice

where p is a proof that ciphertext c has the correct form. Concretely,

p : (Bob **says** (c **isa** (**int for** Alice))).

Proposition constructor **isa** is a built-in constant with the job of witnessing these justified casts.

In combination, true and justified casts allows us to reason about ciphertexts, even those which cannot be decrypted in a particular context. Subject reduction ensures that (for suitable fact contexts) decryption never fails for true-cast ciphertexts. Furthermore, while justified casts may lead to decryption failures, such failures are accompanied by signed **isa** proofs that can be used to assign blame.

Casting allows the programmer to assign a precise type to ciphertext. Conversely, **asbits** strips a ciphertext's annotation, resulting in a term with the following less informative type.

asbits (**cast**($\mathcal{E}$(Alice, 42, 0x2b63)) **to** (**int for** Alice)) : **bits**

```
(* This interface provides a basic API for a network library *)
Signature NetIO

  assert OkToSend: prin → Type → Prop;

  val attempt_acquire_weak_credential:
    (a: prin) → (T: Type) → Maybe (pf (Kernel says OkToSend a T))

  val attempt_acquire_strong_credential:
        (b: prin) →
        Maybe ((a: prin) → (T: Type) → pf (b says OkToSend a T) →
                                        pf (Kernel says OkToSend a T))

  val recv: (T: Type) → T

  val send: (T: Type) → (a: prin) → T →
                pf (Kernel says (OkToSend a T)) → Unit

End Signature
```

Figure 4.1: A simple communications library

Type **bits** classifies naked ciphertexts, and the above term reduces to

$\mathcal{E}$(Alice, 42, 0x2b63) : **bits**.

4.3 Examples

This section shows sample programs that provide intuition for Aura$_{\text{conf}}$. For purposes of illustration, the samples use some features—particularly modules and some trivial type inference—that are not part of the formal language definition.

Figure 4.1 defines a simple networking interface. The functions send and recv are intended to send and receive data values. In addition to data to transmit, send consumes a proof that the system (that is the principal Kernel) permits the operation. Concretely

send **int** Bob 42 p

sends the data value 42 to principal Bob when p is an appropriate access-control proof. Note that both the confidential and non-confidential values may be transmitted over any channel. (This treatment assumes that the system is configured a priori so that each principal is associated with a unique server address ready to accept messages—a possible but not canonical situation.)

Assertion OkToSend and functions attempt_acquire_strong_credential and attempt_acquire_weak_credential define an access control policy for the send function. This first function allows client b to request a proof object permitting arbitrary network writes. The second requests credentials on a transmission-by-transmission basis. The send and receive functions take and return fewer proofs than suggested by Chapter 2's methodology. These streamlined functions may be defined in terms of earlier, fully-annotated versions, passing baked-in constant proofs as needed.

Suppose that a program running with Alice's authority needs to build a secret message that will eventually be read by Bob and perhaps for-bound by Charlie and Daniel.

```
let msg at ⊥=
    let x at Bob = return (int for Bob) 312 in
    let y at ⊥= asbits x in
     cast y to (int for Bob)
            blaming (say Alice (y isa (int for Bob)))
  in
send (int for Bob) Charlie msg (get_cred y)
```

The program takes the annotated ciphertext created for Bob—which will have the form of a true cast—strips its annotation with **asbits**, and creates a justified cast suitable for sending. The true cast is stored in x, whose **at** Bob annotation reflects that the true cast may only be typed in certain contexts—those where Bob's key or relevant facts are available. In contrast, y and msg may be interpreted anywhere; this is reflected by the **at** ⊥ annotations. Finally, get_cred is assumed to return an appropriate access-control proof. Even this simple program relies on the harmonious interaction of several language features: both **cast**s, **asbits**, **return**, **isa**, and **let at**.

Figure 4.2 illustrates a larger program that uses the NetIO interface to implement a storage server. Clients use the NetworkedStore program to store encrypted objects. Each principal has a storage area with a set of slots indexed by natural numbers. A request of form r_put Alice 3 v instructs the server to store value v (of type String **for** Alice) in principal Alice's third storage cell. This value can be retrieved with request r_get Alice 3. The server allows anyone to store or retrieve data from any storage location, even those belonging to another principal. Confidentiality of slot contents is maintained by the use of **for**-types and encryption. It is also possible to add a layer of proof-based access-control to limit access to ciphertexts.

```
Module NetworkedStore Of Server

(* Use the a module which implements the NetIO interface. *)
open NetIOImp: NetIO;

(* The type of network requests to this server *)
data request: Type {
| r_put: (a: prin) → (id: Nat) → String for a → request
| r_get: prin → Nat → request
}

(* The Map datatype stores for each principal a natural-number-indexed
   set of confidential Strings. *)
data Map: Type {
| m_intro : ((a: prin) → Nat → Maybe (String for a)) → Map
}

let empty_map: Map = m_intro (λa: prin. λid: Nat. nothing (String for a)) in

let lookup: Map → (a: prin) → Nat → Maybe (String for a) =
  λm: Map. λa: prin. λn: Nat.
  match m with Maybe (String for a) {
  | m_intro → λf: (a: prin) → Nat → Maybe (String for a). f a n
  }
  in

let insert: Map → (a: prin) → Nat → (String for a) → Map =
  λm: Map. λa: prin. λid: Nat. λmsg: String for a.
    m_intro (λa′: prin. λ id′: Nat.
                  if a = a′
                    then
                      match eqnat id id′ with Maybe (String for a′) {
                      | true → ⟨just (String for a) msg: Maybe(String for a′)⟩
                      | false → lookup m a′ id′
                      }
                    else lookup m a′ id′)
  in


(* A helper function that lets that lets us compose functions of
   type T → S with the for monad. Bind alone only works with
   T → (S for a) functions. *)
let for_lift : (T: Type) → (S: Type) → (a: prin) →
                  (T → S) → T for a → S for a =
  λT: Type. λS: Type. λa: prin. λf: T → S.
    λx: T for a.
      bind y = x
        in (return (f y) as (S for a)) as (S for a)
  in
```

Figure 4.2: Code for confidential storage server

```
(* rewrite_cred uses say and a proof signed by the kernel to produce a new,
   more useful proof access-control proof. *)
let rewrite_cred:
      ((a: prin) → (T: Type) → pf (self says OkToSend a T) →
                                  pf (Kernel says OkToSend a T)) −{self}→
      ((a: prin) → (T: Type) → pf (Kernel says OkToSend a T)) =
  λ{self} p1: ((a: prin) → (T: Type) → pf (self says OkToSend a T) →
                                   pf (Kernel says OkToSend a T)).
    let p2: pf (self says ((b: prin) → (S: Type) → OkToSend b S)) =
              say ((b: prin) → (S: Type) → OkToSend b S) in
    let p3: (b: prin) → (S: Type) → pf (self says (OkToSend b S)) =
           λb:prin. λS:Type.
              bind p2
                  (λp2′:self says ((b:prin) → (S:Type) → OkToSend b S).
                     return (bind p2′
                                (λp2″: (b:prin) → (S:Type) → OkToSend b S.
                                    return self (p2″ b S))))
        in
    λc: prin. λU: Type. p1 c U (p3 c U)
  in

(* attempt_acquire_credential gets a proof allowing access to the send
   function and rewrites it into a useful form using rewrite_cred. *)
let attempt_acquire_credential:
    Unit −{self}→ Maybe ((a: prin) → (T: Type) → pf (Kernel says OkToSend a T)) =
      λ{self} x: Unit.
        match (attempt_acquire_strong_credential self) with
            Maybe ((a: prin) → (T: Type) → pf (Kernel says OkToSend a T)) {
        | just → λ{self} p: (a: prin) → (T: Type) →
                                               pf (self says OkToSend a T) →
                                               pf (Kernel says OkToSend a T).
                      just ((a: prin) → (T: Type) →
                                      pf (Kernel says OkToSend a T))
                          (rewrite_cred p)
        | nothing → nothing ((a: prin) → (T: Type) →
                                     pf (Kernel says OkToSend a T))

        }
  in

(* The main server loop. This reads input requests from the network and
 * stores or retrieves confidential values as needed. *)
let server_loop: ((a: prin) → (T: Type) → pf (Kernel says OkToSend a T)) →
                  Map → Unit =
  λp: (a: prin) → (T: Type) → pf (Kernel says OkToSend a T).
  fun rec: Map → Unit =
    λm: Map. rec
      match recv request with Map {
      | r_put → λa: prin. λid: Nat. λmsg: String for a. insert m a id msg
```

Code for confidential storage server (cont.)

```
100         | r_get  → λa: prin. λid: Nat.
101                     let u: Unit =
102                       match lookup m a id with Unit {
103                       | just  → λmsg: String for a.
104                                     send (String for a) a
105                                           ( for_lift  String String a time_stamp msg)
106
107                                           (p a (String for a))
108                       | nothing → send (String for a) a
109                                           (String a "not found")
110                                           (p a (String for a))
111                       }
112                     in m
113         }
114     in rec end
115   in
116
117 (* This code starts the server loop after acquiring necessary credentials.
118     If such credentials are not available, it fails. *)
119 match attempt_acquire_credential unit with Unit {
120 | just  → λp: ((a: prin) → (T: Type) → pf (Kernel says OkToSend a T)).
121              server_loop p empty_map
122 | nothing → unit
123 }
124
125 End Module
```

Code for confidential storage server (cont.)

The NetworkedStore module is annotated with **Of** Server, indicating that it defines code that will be typechecked and run on behalf of principal Server. In the terminology of Section 4.4, the module's top-level terms must be typechecked with statically available key, soft decryption limit, and effect label all equal to singleton world Server. Furthermore, the code must be run with authority Server.

At the heart of this example is the server_loop function. This reads incoming requests from the network and adds values to, or finds values in, the store. In the case of an r_get request, the loop adds a time stamp to the retrieved value (Line 106). Note that whether or not returned values are timestamped does not affect the type of the resulting object; in general the ability to compose computations with ciphertexts allows the server's behavior to change without breaking existing interfaces. This composition is made possible by function for_lift which is defined using **bind** (Lines 43–48).

As with Section 3.5's music server, the storage server must acquire access-control proofs and rewrite them into useful forms. Function attempt_acquire_credential (Lines 74–89) attempts to get a proof, which permits liberal use of send, from the network module's attempt_acquire_strong_credential function. Success yields a proof with the form of a delegation:

(a: **prin**) → (T: **Type**) → **pf** (Server **says** OkToSend a T) → **pf** (Kernel **says** OkToSend a T).

This is rewritten to a simpler form using rewrite_cred (Lines 53–53). Function rewrite_cred uses **say** to create a fresh (Server **says** ...) proof and compose it with the delegation above. Evaluating **say** requires Server's private key, and this fact is recorded as a latent effect in rewrite_cred's type. The following section discusses effects annotations is more detail.

4.4 Language Definition

This section describes the definition of $\text{Aura}_{\text{conf}}$ and its metatheory.

In type-safe languages such as $\text{Aura}_{\text{conf}}$, a conservative algorithm identifies and rejects programs that might go wrong—that is crash—at runtime. There are many ways that a program can crash, such as by accessing a memory location out of scope or jumping to an invalid instruction sequence. $\text{Aura}_{\text{conf}}$'s type system, like Aura's or ML's, rules out these particular errors. However, an $\text{Aura}_{\text{conf}}$ program could potentially go wrong in several other ways, and the type system must address the following two challenges just to ensure soundness.

Challenge 1 Ensure decryption failures—in which a ciphertext cannot be decrypted to a well-typed plaintext—only occur where a proof can be used to assign blame. Failures without such proofs constitute undefined behavior.

Challenge 2 Ensure that running programs only (attempt to) use private keys that are actually available at runtime. Programs that require unavailable keys for decryption or signing are stuck.

We address the first challenge by constraining the canonical forms of **for** types. Enforcing these constraints requires types and terms to have (loosely) consistent meanings to typecheckers with different capabilities, i.e. different access to private keys. $\text{Aura}_{\text{conf}}$'s type system accomplishes this using ideas based on modal type systems for distributed computing (Jia and Walker, 2004; Murphy et al., 2004; Murphy, 2008).

Worlds			
W, V, U	$::=$	$\bot$	Bottom world (no keys)
	$\mid$	t	Singleton worlds
	$\mid$	$\top$	Top world (all keys)
Terms			
t	$::=$	$\ldots$	(Standard Aura syntax)
	$\mid$	$(x\!:\!t) \rightarrow_{\{W\}} t$	Implication, quantification, arrow
	$\mid$	$\lambda_{\{W\}} x\!:\!t.\, t$	Abstraction
	$\mid$	$a\ \mathbf{for}\ P$	Type of encrypted data
	$\mid$	$\mathbf{return}_f\ e\ \mathbf{as}\ t$	Private computation
	$\mid$	$\mathbf{bind}_f\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{as}\ t$	Composition of private computations
	$\mid$	$\mathbf{run}_f\ e$	Extract private data
	$\mid$	$\mathbf{cast}\ e\ \mathbf{to}\ t$	True case
	$\mid$	$\mathbf{cast}\ e\ \mathbf{to}\ t\ \mathbf{blaming}\ p$	Justified cast
	$\mid$	$\mathbf{say}\ a\ P$	Saying
	$\mid$	$\mathcal{E}(a, e, n)$	Ciphertext
	$\mid$	$\mathbf{fail}\ p$	Decryption-failed exception

Figure 4.3: Aura$_{\text{conf}}$ Syntax

We address the second challenge by statically tracking the use of **say** and **run**, the only operators that use private keys, and ensuring that required keys will be accessible at runtime. To do so, we blend ideas from modal type systems with those from type-and-effect analysis (Lucassen and Gifford, 1988; Talpin and Jouvelot, 1992).

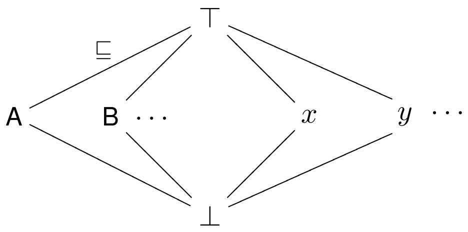
Syntax

Aura$_{\text{conf}}$ adds to and modifies Aura's syntax. These changes are summarized in Figure 4.3, and include the new operators introduced in Section 4.2. To enable the type-and-effect analysis described above, abstractions and arrows are now labeled with *worlds* that summarize latent uses of private keys.

Syntactically, the set of worlds is the set of terms augmented with distinguished top and bottom elements. Aura$_{\text{conf}}$'s static semantics identify only some worlds as well formed: namely principal constants, variables of type **prin**, $\top$, and $\bot$. We define a partial order on worlds,

$$\frac{}{\bot \sqsubseteq W} \qquad\qquad \frac{}{W \sqsubseteq W} \qquad\qquad \frac{}{W \sqsubseteq \top}$$

and can visualize the lattice of well-formed worlds as follows.



Intuitively $W \sqsubseteq U$ when U describes more private keys than W. World $\top$ represents the set of all private keys. Generalizing worlds to arbitrary principal sets would work formally, but is less appealing from a cryptographic perspective.

Unlike in Aura, the **say** operator is now annotated with a principal, so that

$$\textbf{say}\ \mathsf{Alice}\ \mathsf{P} : \mathsf{Alice}\ \textbf{says}\ \mathsf{P}$$

This allows removal of the **self** constant and makes for a consistent treatment of principals throughout Aura$_{\text{conf}}$.

The special term **fail** p represents fatal exceptions caused by decryption failures. Argument p represents a proof to blame for the exception.

Dynamic semantics

The dynamic semantics for Aura$_{\text{conf}}$ makes precise the notion of a program's authority, realistically models the state necessary to perform (pseudo-)randomized cryptography, and enables reasoning about dynamically created ciphertexts.

The evaluation judgment is written

$$\Sigma; \mathcal{F}_0; W \vdash \{\!|e, n|\!\} \ \mapsto \{\!|e', n'|\!\} \text{ learning } \mathcal{F}.$$

This says that an expression e running with W's authority—with the private keys described by world W—steps to e'. Expression e may, as described below, dynamically invoke the type checker, so the evaluation relation contains a signature Σ and fact context $\mathcal{F}_0$ for this purpose. Natural number n represents the initial seed of a randomization vector for encryption; the step updates it to n'. Finally $\mathcal{F}$ is a fact context, with zero or one elements, containing facts about freshly created ciphertexts.

In general Aura$_{\text{conf}}$'s evaluation relation subsumes Aura's. For intuition, when $e \mapsto e'$ in Aura,

$$\Sigma; \mathcal{F}_0; \textbf{self} \vdash \{\!|e, n|\!\} \ \mapsto \{\!|e', n|\!\} \text{ learning } \cdot$$

holds in Aura$_{\text{conf}}$. Figure 4.4 lists the evaluation rules for new operators. (The square-bracket notation in STEP-ASBITS defines a rule that works for both flavors of cast.)

Rules STEP-FORRET and STEP-FORBIND introduce new ciphertexts. In each case the current randomization seed, n, is inserted into the ciphertext and the seed is incremented. Additionally a fact describing the ciphertext is learned. While STEP-FORRET is simple, STEP-FORBIND is more complicated. The latter builds an expression using **let at** that can be run by the destination machine and that performs necessary decryptions.

Rule STEP-FORRUN-OK, STEP-FORRUN-ILLTYPED, and STEP-FORRUN-JUNK attempt to decrypt and typecheck an annotated ciphertext, signaling an error as needed.

Figure 4.4 elides several congruence rules. They are all similar to STEP-APP-CONGL, which copies its premise's new facts and randomization seed.

Finally, we assume newly generated facts are recorded for future typechecking. It is tempting omit fact generation and collection from an implementation, and a type-erasing compiler could do so while retaining a weakened type soundness property. (Preservation would not hold). Such an implementation would be unable to recheck ciphertexts it had created, sent to a third-party, and received again. Balancing improved execution speed against lost expressivity is perhaps best left to users—an implementation could allow compilation both with and without recording new facts.

Static Semantics

Aura$_{\text{conf}}$'s static semantics is based on Aura's, but with several substantial changes. Aura$_{\text{conf}}$ typechecks programs using the following judgments.

Well-formed signature	$\Sigma \vdash \diamond$
Well-formed typing environment	$\Sigma; \mathcal{F}; W \mid \; \vdash E$
Well-formed world (V)	$\Sigma; \mathcal{F}; W \mid E; V \vdash \diamond$
Well-formed worlds (V and U)	$\Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond$
Well-typed term	$\Sigma; \mathcal{F}; W \mid E; V; U \vdash t : s$
Well-typed match branches	$\Sigma; \mathcal{F}; W \mid E; V; U; s; \mathit{args} \vdash \mathit{branches} : t$

The typing relation (well-typed term) has grown substantially. How can we read this judgment?

$$\boxed{\Sigma; F_0; W \vdash \{\!| e_1, n_1 |\!\} \ \mapsto \{\!| e_2, n_2 |\!\} \text{ learning } F}$$

STEP-LETAT

$$\frac{\textit{val } v}{\Sigma; F_0; W \vdash \{\!| \textbf{let } x \textbf{ at } V = v \textbf{ in } e, n |\!\} \ \mapsto \{\!| \{v/x\}e, n |\!\} \text{ learning } \cdot}$$

STEP-FORRET

$$\frac{}{\begin{array}{r}\Sigma; F_0; W \vdash \{\!| \textbf{return}_f\ e \textbf{ as } (t \textbf{ for } a), n |\!\} \mapsto \{\!| \textbf{cast } \mathcal{E}(a, e, n) \textbf{ to } (t \textbf{ for } a), n+1 |\!\} \\ \text{learning } \mathcal{E}(a, e, n) : t \textbf{ for } a \qquad\end{array}}$$

STEP-FORBIND

$$\frac{\textit{val } v}{\begin{array}{l}\Sigma; F_0; W \vdash \{\!| \textbf{bind}_f\ x = v \textbf{ in } e \textbf{ as } t \textbf{ for } a, n |\!\} \\ \qquad\qquad \mapsto \{\!| \textbf{cast } \mathcal{E}(a, \textbf{let } x \textbf{ at } a = (\textbf{run}_f\ v) \textbf{ in } (\textbf{run}_f\ e), n) \textbf{ to } (t \textbf{ for } a), n+1 |\!\} \\ \qquad\qquad \text{learning } \mathcal{E}(a, \textbf{let } x \textbf{ at } a = (\textbf{run}_f\ v) \textbf{ in } (\textbf{run}_f\ e), n) : t \textbf{ for } a\end{array}}$$

STEP-ASBITS

$$\frac{\textit{val}(\textbf{cast } \mathcal{E}(a, e, m) \textbf{ to } t \ [\ \textbf{blaming } p\ ])}{\Sigma; F_0; W \vdash \{\!| \textbf{asbits cast } \mathcal{E}(a, e, m) \textbf{ to } t \ [\ \textbf{blaming } p\ ], n |\!\} \ \mapsto \{\!| \mathcal{E}(a, e, m), n |\!\} \text{ learning } \cdot}$$

STEP-FORRUN-OK

$$\frac{\textit{val}(\textbf{cast } \mathcal{E}(a, e, m) \textbf{ to } t \ [\ \textbf{blaming } p\ ]) \qquad \Sigma; F_0; W \mid \cdot; a; a \vdash e : t \qquad a \sqsubseteq W}{\Sigma; F_0; W \vdash \{\!| \textbf{run}_f\ (\textbf{cast } \mathcal{E}(a, e, m) \textbf{ to } t \ [\ \textbf{blaming } p\ ]), n |\!\} \ \mapsto \{\!| e, n |\!\} \text{ learning } \cdot}$$

STEP-FORRUN-ILLTYPED

$$\frac{\textit{val}(\textbf{cast } \mathcal{E}(a, e, m) \textbf{ to } t \textbf{ blaming } p) \qquad \Sigma; F_0; W \mid \cdot; a; a \nvdash e : t \qquad a \sqsubseteq W}{\Sigma; F_0; W \vdash \{\!| \textbf{run}_f\ (\textbf{cast } \mathcal{E}(a, e, m) \textbf{ to } t \textbf{ blaming } p), n |\!\} \ \mapsto \{\!| \textbf{fail } p, n |\!\} \text{ learning } \cdot}$$

STEP-FORRUN-JUNK

$$\frac{\textit{val}(\textbf{cast } \mathcal{E}(a, e, m) \textbf{ to } (t \textbf{ for } b) \textbf{ blaming } p) \qquad b \sqsubseteq W \qquad a \neq b}{\Sigma; F_0; W \vdash \{\!| \textbf{run}_f\ (\textbf{cast } \mathcal{E}(a, e, m) \textbf{ to } (t \textbf{ for } b) \textbf{ blaming } p), n |\!\} \ \mapsto \{\!| \textbf{fail } p, n |\!\} \text{ learning } \cdot}$$

STEP-APP-CONGL

$$\frac{\Sigma; F_0; W \vdash \{\!| e_1, n |\!\} \ \mapsto \{\!| e_1', n' |\!\} \text{ learning } F}{\Sigma; F_0; W \vdash \{\!| e_1\ e_2, n |\!\} \ \mapsto \{\!| e_1'\ e_2, n' |\!\} \text{ learning } F}$$

Figure 4.4: Selected Aura$_{\text{conf}}$ evaluation rules

Facts, worlds, and the typing judgment

Meta-variable $\mathcal{F}$ is a fact context as described in Section 4.2. It's formally defined by the grammar

$$\begin{array}{lcl} & \text{Fact Contexts} & \\ \mathcal{F} & ::= & \cdot \mid \mathcal{F}, \mathcal{E}(a, e, n) : t. \end{array}$$

Intuitively, typechecking uses the fact context to associate typing information with newly created ciphertexts. This is important, because ciphertexts are not generally amenable to inspection.

World W, the *statically available key*, describes which key is available to the typechecker at compile time. As in the dynamic semantics, W represents the key that is currently on hand, ready for use. We will only consider singleton and bottom worlds here; typechecking a program with $W = \top$ corresponds to having all private keys available at once—a well-defined but unlikely scenario. Together the fact context and statically available key determine a hard limit on the typechecker's ability to reason about ciphertext.

World V, the *soft decryption limit*, is a formal upper limit bounding which decryption keys or facts should be used when typechecking a particular term. Intuitively the keys used to typecheck a term are $W \sqcap V$. The soft decryption limit is necessary to deal with mobile code. Consider what happens when Alice creates a string **for** Bob. She is building an object containing a subterm, say s, that Bob must decrypt and typecheck as string. However, Bob must check s without the benefit of Alice's private key and using a different fact context. (Because s might be a computation containing nested binds, Bob's task is non-trivial.) To account for this, Alice's typing derivation uses $V = \mathsf{Bob}$ when checking s, thus ensuring Bob can understand s without Alice's private information or state. Typechecking a top-level program takes place with $V = \top$, indicating no restriction on key or fact use.

The interaction between fact context, statically available key and soft decryption limit can be better understood by examining simplified versions of typing rules WF-TM-FORRET, WF-TM-CASTDEC, and WF-TM-CASTFACT. WF-TM-FORRET has form

$$\frac{\_; \mathcal{F}; W \mid \_; a; \_ \vdash e : t \qquad a \sqsubseteq V \qquad \cdots}{\_; \mathcal{F}; W \mid \_; V; \_ \vdash \textbf{return}_f\ e\ \textbf{as}\ (t\ \textbf{for}\ a) : t\ \textbf{for}\ a} \ .$$

This rule is packaging expression e for consumption by principal a. The first premise checks that e is classified by t under soft decryption limit a—this will ensure that the derivation will work even when a does not have access to the facts in $\mathcal{F}$ or a private key indicated by W. Having checked e under this restriction it's ok to conclude that $\textsf{\textbf{return}}_f\ e\ \textsf{\textbf{as}}\ (t\ \textsf{\textbf{for}}\ a)$ has type $t\ \textsf{\textbf{for}}\ a$ in a less restricted context, where soft decryption limit V contains a's key (i.e., $a \sqsubseteq V$).

Observe that elements right of the vertical bar differed between WF-TM-FORRET's premise and conclusion, but symbols on the bar's left stayed the same. In general, symbols left of the bar are parameters of the relation and are held constant throughout an entire derivation tree. Symbols right of the bar are indices and may change within a derivation.

All typing rules in this section are simplified. Unabridged versions of these and other key typing rules are written in standard notation in Figure 4.5. Appendix C gives all Aura$_{\text{conf}}$ definitions in Coq notation.

The statically available key is used directly in rule WF-TM-CASTDEC.

$$\frac{b \sqsubseteq W \qquad b \sqsubseteq V \qquad \_; \mathcal{F}; W \mid \_; V; \_ \vdash e : t}{\_; \mathcal{F}; W \mid \_; V; \_ \vdash \textsf{\textbf{cast}}\ \mathcal{E}(b, e, n)\ \textsf{\textbf{to}}\ (t\ \textsf{\textbf{for}}\ b) : t\ \textsf{\textbf{for}}\ b}$$

Here an annotated ciphertext encrypted for b is type checked by decrypting and recursively type-checking its contents. Premise $b \sqsubseteq W$ shows the statically available key is sufficient to perform the decryption. WF-TM-CASTDEC may only be applied when current soft decryption limit V is greater than b. This last point is important. Together with setting the soft decryption limit to a in WF-TM-FORRET, it ensures that impossible decryptions are not required to later typecheck data packaged by correct programs.

Finally, WF-TM-CASTFACT has form

$$\frac{(\mathcal{E}(a, e, n) : t\ \textsf{\textbf{for}}\ b) \in F \qquad b \sqsubseteq V}{\_; \mathcal{F}; W \mid \_; V; \_ \vdash \textsf{\textbf{cast}}\ \mathcal{E}(a, e, n)\ \textsf{\textbf{to}}\ (t\ \textsf{\textbf{for}}\ b) : t\ \textsf{\textbf{for}}\ b} \ .$$

This says that an annotated piece of ciphertext can have type $t\ \textsf{\textbf{for}}\ b$ when a fact indicates the type. As above, soft decryption limit V must be greater than b.

Aura$_{\text{conf}}$ typing contexts track a soft decryption limit for each bound variable. This is necessary to ensure that the substitution property—replacing variables with appropriate values maintains a term's type—can be stated precisely (Lemma 10). Formally, Aura$_{\text{conf}}$ environments are defined by

$\boxed{\Sigma;\mathcal{F};W \mid E;V;U \vdash e : t}$

$$\frac{\Sigma;\mathcal{F};W \mid E;V;U \vdash \diamond}{\Sigma;\mathcal{F};W \mid E;V;U \vdash \textsf{\textbf{Type}} : \textsf{\textbf{Kind}}}\;\textsc{wf-tm-Type}$$

$$\frac{\begin{array}{c}\Sigma;\mathcal{F};W \mid E;\bot;\bot \vdash u_1 : k_1 \qquad \Sigma;\mathcal{F};W \mid E;V;U \vdash \diamond \\ \Sigma;\mathcal{F};W \mid E;V;U_0 \vdash \diamond \qquad \Sigma;\mathcal{F};W \mid E, x : u_1\,\textsf{\textbf{at}}\,\bot;\bot;\bot \vdash u_2 : k_2 \\ k_2 \in \{\textsf{\textbf{Type}},\textsf{\textbf{Prop}},\textsf{\textbf{Kind}}\} \qquad k_1 \in \{\textsf{\textbf{Type}},\textsf{\textbf{Prop}}\} \vee u_1 \in \{\textsf{\textbf{Type}},\textsf{\textbf{Prop}}\}\end{array}}{\Sigma;\mathcal{F};W \mid E;V;U \vdash (x{:}u_1) \rightarrow_{\{U_0\}} u_2 : k_2}\;\textsc{wf-tm-Arr}$$

$$\frac{\Sigma;\mathcal{F};W \mid E;V;U \vdash \diamond \qquad (x : t\,\textsf{\textbf{at}}\,V_0) \in E \qquad V_0 \sqsubseteq V}{\Sigma;\mathcal{F};W \mid E;V;U \vdash x : t}\;\textsc{wf-tm-Var}$$

$$\frac{\begin{array}{c}\Sigma;\mathcal{F};W \mid E;\bot;\bot \vdash u_1 : k_1 \\ \Sigma;\mathcal{F};W \mid E;V;U_0 \vdash \diamond \qquad \Sigma;\mathcal{F};W \mid E, x : u_1\,\textsf{\textbf{at}}\,\bot;V;U_0 \vdash f : u_2 \\ \Sigma;\mathcal{F};W \mid E;V;U \vdash (x{:}u_1) \rightarrow_{\{U_0\}} u_2 : k \\ k \in \{\textsf{\textbf{Type}},\textsf{\textbf{Prop}}\} \qquad k_1 \in \{\textsf{\textbf{Type}},\textsf{\textbf{Prop}}\} \vee u_1 \in \{\textsf{\textbf{Type}},\textsf{\textbf{Prop}}\}\end{array}}{\Sigma;\mathcal{F};W \mid E;V;U \vdash \lambda_{\{U_0\}} x{:}u_1.\; f : (x{:}u_1) \rightarrow_{\{U_0\}} u_2}\;\textsc{wf-tm-Abs}$$

$$\frac{\begin{array}{c}\Sigma;\mathcal{F};W \mid E;V;U \vdash e_1 : (x{:}t_2) \rightarrow_{\{U_0\}} u \qquad \Sigma;\mathcal{F};W \mid E;\bot;U_2 \vdash e_2 : t_2 \\ \Sigma;\mathcal{F};W \mid E;V;U \vdash t_2 : k_2 \qquad \Sigma;\mathcal{F};W \mid E;V;U \vdash \{e_2/x\}u : k_u \qquad U_0 \sqsubseteq U \\ \left(\begin{array}{c}(\mathit{val}\;e_2 \wedge U_2 = \bot) \vee (k_u = \textsf{\textbf{Type}} \wedge x \notin \mathit{fv}(u) \wedge U_2 = U) \\ \vee\,(k_2 \in \{\textsf{\textbf{Prop}},\textsf{\textbf{Kind}}\} \wedge x \notin \mathit{fv}(u) \wedge U_2 = U)\end{array}\right)\end{array}}{\Sigma;\mathcal{F};W \mid E;V;U \vdash e_1\;e_2 : \{x/e_2\}u}\;\textsc{wf-tm-App}$$

$$\frac{\begin{array}{c}\Sigma;\mathcal{F};W \mid E;V_1;U \vdash e_1 : t_1 \\ \Sigma;\mathcal{F};W \mid E;V;V_1 \vdash \diamond \qquad \Sigma;\mathcal{F};W \mid E;\bot;\bot \vdash t_1 : k \\ \Sigma;\mathcal{F};W \mid E, x : t_1\,\textsf{\textbf{at}}\,V_1;V;U \vdash e_2 : t \qquad k \in \{\textsf{\textbf{Type}},\textsf{\textbf{Prop}}\} \qquad V_1 \sqsubseteq V\end{array}}{\Sigma;\mathcal{F};W \mid E;V;U \vdash \textsf{\textbf{let}}\;x\,\textsf{\textbf{at}}\,V_1 = e_1\;\textsf{\textbf{in}}\;e_2 : t}\;\textsc{wf-tm-LetAt}$$

$$\frac{\begin{array}{c}\Sigma;\mathcal{F};W \mid E;b;U \vdash e_1 : t_1\,\textsf{\textbf{for}}\,b \qquad \Sigma;\mathcal{F};W \mid E;b \vdash \diamond \\ \Sigma;\mathcal{F};W \mid E;\bot;\bot \vdash t_1 : \textsf{\textbf{Type}} \qquad \Sigma;\mathcal{F};W \mid E;\bot;\bot \vdash t\,\textsf{\textbf{for}}\,b : \textsf{\textbf{Type}} \\ \Sigma;\mathcal{F};W \mid E;V \vdash \diamond \qquad \Sigma;\mathcal{F};W \mid E, x : t_1\,\textsf{\textbf{at}}\,b;b;b \vdash e_2 : t\,\textsf{\textbf{for}}\,b \qquad b \sqsubseteq V\end{array}}{\Sigma;\mathcal{F};W \mid E;V;U \vdash \textsf{\textbf{bind}}_f\;x = e_1\;\textsf{\textbf{in}}\;e_2\;\textsf{\textbf{as}}\;(t\,\textsf{\textbf{for}}\,b) : t\,\textsf{\textbf{for}}\,b}\;\textsc{wf-tm-ForBind}$$

$$\frac{\begin{array}{c}\mathit{val}\;a \qquad \Sigma;\mathcal{F};W \mid \cdot;\bot;\bot \vdash a : \textsf{\textbf{prin}} \\ \Sigma;\mathcal{F};W \mid \cdot;\bot;\bot \vdash P : \textsf{\textbf{Prop}} \qquad \Sigma;\mathcal{F};W \mid E;V;U \vdash P : \textsf{\textbf{Prop}}\end{array}}{\Sigma;\mathcal{F};W \mid E;V;U \vdash \textsf{\textbf{sign}}(a,P) : a\,\textsf{\textbf{says}}\,P}\;\textsc{wf-tm-Sign}$$

$$\frac{\Sigma;\mathcal{F};W \mid E;V;U \vdash \diamond}{\Sigma;\mathcal{F};W \mid E;V;U \vdash \textsf{\textbf{prin}} : \textsf{\textbf{Type}}}\;\textsc{wf-tm-Prin} \qquad \frac{\Sigma;\mathcal{F};W \mid E;V;U \vdash \diamond}{\Sigma;\mathcal{F};W \mid E;V;U \vdash \textsf{\textbf{bits}} : \textsf{\textbf{Type}}}\;\textsc{wf-tm-Bits}$$

Figure 4.5: Selected typing rules for Aura$_{\text{conf}}$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W \mid E; \bot; \bot \vdash P : \textsf{\textbf{Prop}} \\ \Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond \qquad \Sigma; \mathcal{F}; W \mid E; \bot; \bot \vdash a : \textsf{\textbf{prin}} \qquad a \sqsubseteq U \qquad \mathit{val}\ a\end{array}}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \textsf{\textbf{say}}\, a\, P : \textsf{\textbf{pf}}\ (a\, \textsf{\textbf{says}}\, P)}\ \textsc{wf-tm-Say}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W \mid E; V; \bot \vdash t : \textsf{\textbf{Type}} \\ \Sigma; \mathcal{F}; W \mid E; V; \bot \vdash a : \textsf{\textbf{prin}} \qquad \Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond \qquad \mathit{val}\ t \qquad \mathit{val}\ a\end{array}}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash t\, \textsf{\textbf{for}}\, a : \textsf{\textbf{Type}}}\ \textsc{wf-tm-For}$$

$$\frac{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \mathcal{E}(t_1, t_2, n) : \textsf{\textbf{bits}}}\ \textsc{wf-tm-Enc}$$

$$\frac{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond \qquad E \mid\!\!\sim e : t\, \textsf{\textbf{for}}\, a}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \textsf{\textbf{asbits}}\ e : \textsf{\textbf{bits}}}\ \textsc{wf-tm-AsBits}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W \mid E; a; a \vdash e : t \\ \Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond \qquad \Sigma; \mathcal{F}; W \mid E; \bot; \bot \vdash t\, \textsf{\textbf{for}}\, a : \textsf{\textbf{Type}} \qquad a \sqsubseteq V\end{array}}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \textsf{\textbf{return}}_f\ e\ \textsf{\textbf{as}}\ (t\, \textsf{\textbf{for}}\, a) : t\, \textsf{\textbf{for}}\, a}\ \textsc{wf-tm-ForRet}$$

$$\frac{\Sigma; \mathcal{F}; W \mid E; V; U \vdash t\, \textsf{\textbf{for}}\, b : \textsf{\textbf{Type}} \qquad \Sigma; \mathcal{F}; W \mid E; V; U \vdash e : \textsf{\textbf{bits}} \qquad \mathit{val}\ e}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash e\, \textsf{\textbf{isa}}\, t\, \textsf{\textbf{for}}\, b : \textsf{\textbf{Prop}}}\ \textsc{wf-tm-Isa}$$

$$\frac{\Sigma; \mathcal{F}; W \mid E; V; U \vdash e : t\, \textsf{\textbf{for}}\, a \qquad a \sqsubseteq V \qquad a \sqsubseteq U}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \textsf{\textbf{run}}_f\ e : t}\ \textsc{wf-tm-ForRun}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W \mid E; V; U \vdash e : t_1 \\ \Sigma; \mathcal{F}; W \mid E; V; U \vdash t_2 : \textsf{\textbf{Type}} \qquad \mathit{converts}\ E\ t_1\ t_2\ \mathit{at}\ V\end{array}}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \langle e : t_2 \rangle : t_2}\ \textsc{wf-tm-CastConv}$$

$$\frac{\begin{array}{c}(\mathcal{E}(a, e, n) : t\, \textsf{\textbf{for}}\, b) \in \mathcal{F} \\ \Sigma; \mathcal{F}; W \mid E; \bot; \bot \vdash t\, \textsf{\textbf{for}}\, b : \textsf{\textbf{Type}} \qquad \Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond \qquad b \sqsubseteq V\end{array}}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \textsf{\textbf{cast}}\ \mathcal{E}(a, e, n)\ \textsf{\textbf{to}}\ (t\, \textsf{\textbf{for}}\, b) : t\, \textsf{\textbf{for}}\, b}\ \textsc{wf-tm-CastFact}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W \mid \cdot; b; b \vdash e : t \qquad \Sigma; \mathcal{F}; W \mid \cdot; \bot; \bot \vdash t\, \textsf{\textbf{for}}\, b : \textsf{\textbf{Type}} \\ \Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond \qquad \Sigma; \mathcal{F}; W \mid \cdot; b \vdash \diamond \qquad b \sqsubseteq V \qquad b \sqsubseteq W\end{array}}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \textsf{\textbf{cast}}\ \mathcal{E}(b, e, n)\ \textsf{\textbf{to}}\ (t\, \textsf{\textbf{for}}\, b) : t\, \textsf{\textbf{for}}\, b}\ \textsc{wf-tm-CastDec}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W \mid E; V; U \vdash p : \textsf{\textbf{pf}}\ (a\, \textsf{\textbf{says}}\, (e\, \textsf{\textbf{isa}}\, t\, \textsf{\textbf{for}}\, b)) \\ \Sigma; \mathcal{F}; W \mid E; V; U \vdash e : \textsf{\textbf{bits}} \qquad \Sigma; \mathcal{F}; W \mid E; \bot; \bot \vdash t\, \textsf{\textbf{for}}\, b : \textsf{\textbf{Type}} \qquad \mathit{val}\ e\end{array}}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \textsf{\textbf{cast}}\ e\ \textsf{\textbf{to}}\ (t\, \textsf{\textbf{for}}\, b)\ \textsf{\textbf{blaming}}\ p : t\, \textsf{\textbf{for}}\, b}\ \textsc{wf-tm-CastJust}$$

Selected typing rules for Aura$_{\text{conf}}$ (cont.)

the following grammar.

$$\begin{array}{lll} & \text{Environments} & \\ E & ::= & \cdot \mid E, x : t \textsf{ at } W \mid E, x \sim (t_1 = t_2) : u \textsf{ at } W \end{array}$$

Equalities in the environment are also annotated with worlds. At present this mainly provides uniformity but also allows for language extensions that consider local equalities.

The typing relation's final new metavariable, U, is the judgment's *effect label*. This summarizes the keys that are necessary to successfully execute a piece of code. Effect label $U = \bot$ indicates that an expression is *pure*—that it can execute with no private keys. For instance, typing **say** looks like

$$\frac{\Sigma; \mathcal{F}; W \mid E; \bot; \bot \vdash P : \textsf{Prop} \qquad \Sigma; \mathcal{F}; W \mid E; \bot; \bot \vdash a : \textsf{prin} \qquad a \sqsubseteq U \qquad \cdots}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \textsf{say}\, a\, P : \textsf{pf}\ a\, \textsf{says}\, P} \, .$$

Operator **say** needs to sign P with a's private key, and premise $a \sqsubseteq U$ records that fact. Additionally Aura$_{\text{conf}}$ typing maintains the invariant that type-level terms, such as a **says** P, are pure. Checking the rule's premises with bottom effect label helps to enforce this condition.

It's important to understand the distinction between a judgment's soft decryption limit and effect label. The soft decryption limit controls access to a private key used *statically* for type checking. In contrast, the effect label describes keys used *dynamically* for decryption and signing. It's appealing to attempt to conflate these, but my attempts to do so were imprecise, inelegant, or plain incorrect. The difficulties arise from several considerations. Consider the application $f(\lambda x.e)$ where f does not apply $\lambda x.e$. We want the type system to require a sufficient soft decryption limit to analyze e's embedded ciphertexts. In contrast, e's latent effects will never be forced we would like the application to check with bottom effect label. It's unclear how a single annotation can accommodate both views; using a separate soft decryption limit and effect label resolves this tension. More generally, the type system treats soft decryption limits like Jia and Walker's (2004) **at** modality, while the effect labels are inspired by standard type-and-effect systems. Technically, these analyses are quite different and it's unsurprising that to reap the benefits of both requires incorporating mechanisms inspired by each.

Auxiliary judgments

Aura$_{\text{conf}}$'s well-formed signature judgment is a translation of Aura's, accounting for the new arrow syntax and the typing judgment's new shape. All types in a signature are required to typecheck in the empty fact context and with each of the statically available key, soft decryption limit, and effect label as bottom

The well-typed match branches judgment is updated to deal with effects in the natural way. A list of branches checks under effect label U when each constituent branch also checks under U. Note that, by (a generalization of) Lemma 8, this is equivalent to letting branches check with various effect labels, each less than U.

The formal definitions of the foregoing relations are elided. The Coq code mechanizing the definitions is given in Appendix C. Additionally, these rules may be readily reconstructed from the corresponding definitions for Aura, written in standard notation in Appendix B.

Definitions of environment, world, and worlds well-formedness are more novel and are detailed in Figure 4.6.

The well-formed environment relation checks that all world annotations are themselves well-formed. Additionally, type-level variables (i.e., those classified by **Type** or **Prop**) may only be annotated with world $\bot$. The well-formed environment relation also ensures that the statically available key is a *simple world*—either a principal constant or $\bot$. Intuitively this ensures statically available keys are actual keys, not just variables.

The well-formed world relation always accepts $\top$ and $\bot$. If the world wraps a term, it must be value of type **prin**. The well-formed worlds relation checks that two worlds, typically representing a soft decryption limit and effect label, are well-formed.

New and modified language constructs

Moving from Aura to Aura$_{\text{conf}}$ requires broad changes to the static semantics. Here we will examine the most interesting aspects of the new static semantics, using simplified typing rules. Again, full versions of these rules are printed in Figure 4.5.

$$\boxed{\mathit{simple}\ W}$$

$$\frac{}{\mathit{simple}\ \bot}\ \textsc{sw-Bot} \qquad\qquad \frac{a \in \{\mathsf{A}, \mathsf{B}, \mathsf{C} \ldots\}}{\mathit{simple}\ a}\ \textsc{sw-PrinConst}$$

$$\boxed{\Sigma; \mathcal{F}; W \mid \;\vdash E}$$

$$\frac{\mathit{simple}\ W}{\Sigma; \mathcal{F}; W \mid \;\vdash \cdot}\ \textsc{wf-env-Nil}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W \mid \;\vdash E \qquad \Sigma; \mathcal{F}; W \mid E; V \vdash \diamond \qquad \Sigma; \mathcal{F}; W \mid E; \bot; \bot \vdash t : k \\ x \text{ fresh} \qquad (k \in \{\textbf{Type}, \textbf{Prop}\}) \vee (t \in \{\textbf{Type}, \textbf{Prop}\} \wedge V = \bot)\end{array}}{\Sigma; \mathcal{F}; W \mid \;\vdash E, x : t \,\textbf{at}\, V}\ \textsc{wf-env-ConsVar}$$

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W \mid \;\vdash E \qquad \Sigma; \mathcal{F}; W \mid E; \bot; U \vdash e_1 : t \\ \Sigma; \mathcal{F}; W \mid E; \bot; U \vdash e_2 : t \qquad \mathit{atomic}\, \Sigma\, t \qquad x \text{ fresh} \\ \mathit{val}\, e_1 \qquad \mathit{val}\, e_2 \qquad \Sigma; \mathcal{F}; W \mid E; \bot; \bot \vdash t : \textbf{Type} \qquad \Sigma; \mathcal{F}; W \mid E; V \vdash \diamond\end{array}}{\Sigma; \mathcal{F}; W \mid \;\vdash E, x \sim (e_1 = e_2) : t \,\textbf{at}\, V}\ \textsc{wf-env-ConsEq}$$

$$\boxed{\Sigma; \mathcal{F}; W \mid E; V \vdash \diamond}$$

$$\frac{\Sigma; \mathcal{F}; W \mid \;\vdash E}{\Sigma; \mathcal{F}; W \mid E; \bot \vdash \diamond}\ \textsc{wf-world-Bot}$$

$$\frac{\Sigma; \mathcal{F}; W \mid E; \bot; \bot \vdash a : \textbf{prin} \qquad \mathit{val}\ a}{\Sigma; \mathcal{F}; W \mid E; a \vdash \diamond}\ \textsc{wf-world-Prin}$$

$$\frac{\Sigma; \mathcal{F}; W \mid \;\vdash E}{\Sigma; \mathcal{F}; W \mid E; \top \vdash \diamond}\ \textsc{wf-world-Top}$$

$$\boxed{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond}$$

$$\frac{\Sigma; \mathcal{F}; W \mid E; V \vdash \diamond \qquad \Sigma; \mathcal{F}; W \mid E; U \vdash \diamond}{\Sigma; \mathcal{F}; W \mid E; V; U \vdash \diamond}\ \textsc{wf-worlds}$$

Figure 4.6: Major auxiliary judgments for Aura$_{\text{conf}}$'s static semantics

Variables and binding with soft decryption limits

Application, abstraction, and variable expressions are changed when moving from Aura to Aura$_{\text{conf}}$. This is necessary to work with soft decryption limits and effect labels.

The variable rule is

$$\frac{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash \diamond \qquad (x : t \,\textsf{at}\, V_0) \in E \qquad V_0 \sqsubseteq V}{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash x : t} \quad .$$

From an Aura$_{\text{conf}}$ perspective the important part is the premise $V_0 \sqsubseteq V$. Elsewhere, we ensure that whenever some value v is substituted for x that value is well typed with soft decryption limit V_0. Lemmas 8 and 9 ensure that v will typecheck under x's soft decryption limit and effect label, V and U.

A function's type is annotated with its body's suspended effects. The typing rule looks like

$$\frac{\Sigma; \mathcal{F}; W \,|\, E; V; U_0 \vdash \diamond \qquad \Sigma; \mathcal{F}; W \,|\, E, x : u_1 \,\textsf{at}\, \bot; V; U_0 \vdash f : u_2 \qquad \cdots}{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash \lambda_{\{U_0\}} x{:}u_1.\, f : (x{:}u_1) \rightarrow_{\{U_0\}} u_2} \quad .$$

The rule could be generalized by allowing latent effect label U_0 to depend on x. This was omitted in the interest of simplicity. Dependent effects can still be written; they must reference variables quantified at a surrounding abstraction. To avoid annotating every abstraction with a soft decryption limit, this rule binds x at bottom.

Abstractions are used at applications. The essence of application typing is as follows.

$$\frac{\begin{array}{c}\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash e_1 : (x{:}t_2) \rightarrow_{\{U_0\}} u \qquad \Sigma; \mathcal{F}; W \,|\, E; \bot; U_2 \vdash e_2 : t_2 \\ (\mathit{val}\; e_2 \wedge U_2 = \bot) \vee (x \notin \mathit{fv}(u) \wedge U_2 = U) \qquad U_0 \sqsubseteq U \qquad \cdots\end{array}}{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash e_1\; e_2 : \{x/e_2\}u}$$

Application ensures that argument e_2 is typeable with bottom soft decryption limit; this matches with abstraction typing. Because evaluating the abstraction may trigger latent effect U_0, we require $U_0 \sqsubseteq U$. When e_2 is not a value—which implies e_1's type is not dependent—e_2 may also have have an effect label up to U.

So far we've only seen a way to introduce variables **at** $\bot$. The **let at** construct allows us to reason about variables with different soft decryption limits. This construct's typing rule is summarized by

$$\frac{\Sigma; \mathcal{F}; W \,|\, E; V_1; U \vdash e_1 : t_1 \qquad \Sigma; \mathcal{F}; W \,|\, E, x : t_1 \,\textsf{at}\, V_1; V; U \vdash e_2 : t \qquad V_1 \sqsubseteq V \qquad \cdots}{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash \textsf{let}\ x\ \textsf{at}\ V_1 = e_1\ \textsf{in}\ e_2 : t} \quad .$$

Here e_1 is checked with soft decryption limit V_1 and is bound to x in e_2. In e_2's environment, x is typed **at** V_1. The restriction $V_1 \sqsubseteq V$ is necessary to prevent **let at**s from raising the soft decryption limit and allowing the unsafe use of facts or statically available keys. While **let at** could be defined as a derived form, based on an enhanced version of abstraction, the independent construct simplifies function definition and breaks the language into simple, orthogonal pieces.

The ciphertext and the for monad

The Aura$_{\text{conf}}$ type system always interprets unannotated ciphertexts as unintelligible blobs.

$$\frac{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash \diamond}{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash \mathcal{E}(t_1, t_2, n) : \textsf{bits}}$$

As discussed above, more precise typings maybe given to ciphertexts annotated with true casts or justified casts.

The main operators for working with confidential values are **return**, **run**, and **bind**. The **return** operator packages an expression as a confidential computation and is typed as follows.

$$\frac{\Sigma; \mathcal{F}; W \,|\, E; a; a \vdash e : t \qquad a \sqsubseteq V \qquad \cdots}{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash \textsf{return}_f\ e\ \textsf{as}\ (t\,\textsf{for}\,a) : t\,\textsf{for}\,a} \; .$$

Because e will eventually be run with a's authority it is type checked with soft decryption limit and effect label a. Typically $W \not\sqsubseteq a$ so setting the soft decryption limit to a prevents statically available key W from being used when checking e—important because W will not be on hand when a's program needs to check e. Likewise effect label a rules out inappropriate occurrences of **say** or **run**$_f$. Typing for **bind**$_f$ works analogously; see rule WF-TM-FORBIND.

The **run**$_f$ operator decrypts and evaluates annotated ciphertexts. It's typed by:

$$\frac{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash e : t\,\textsf{for}\,a \qquad a \sqsubseteq V \qquad a \sqsubseteq U}{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash \textsf{run}_f\ e : t}$$

$$\boxed{E \mathrel{|\!\sim} e : t}$$

$$\frac{(x : t\;\textsf{\textbf{at}}\;V) \in E}{E \mathrel{|\!\sim} x : t}\;\textsc{ge-tm-Var} \qquad\qquad \frac{}{E \mathrel{|\!\sim} \mathcal{E}(a, e, n) : \textsf{\textbf{bits}}}\;\textsc{ge-tm-Enc}$$

$$\frac{\begin{array}{c} \mathit{val}\;\textsf{\textbf{cast}}\;e\;\textsf{\textbf{to}}\;t\;\textsf{\textbf{for}}\;a\;[\;\textsf{\textbf{blaming}}\;p\;] \\ E \mathrel{|\!\sim} e : \textsf{\textbf{bits}} \qquad \forall x \in \mathrm{vars}(p) \cup \mathrm{vars}(t\;\textsf{\textbf{for}}\;a).\exists t_x, V_x.(x : t_x\;\textsf{\textbf{at}}\;V_x) \in E \end{array}}{E \mathrel{|\!\sim} \textsf{\textbf{cast}}\;e\;\textsf{\textbf{to}}\;t\;\textsf{\textbf{for}}\;a\;[\;\textsf{\textbf{blaming}}\;p\;] : t\;\textsf{\textbf{for}}\;a}\;\textsc{ge-tm-Cast}$$

Figure 4.7: Approximate typing judgment used by WF-TM-ASBITS

Premise $a \sqsubseteq U$ forces effect label U to record that the **run**$_f$ uses a's private key. Premise $a \sqsubseteq V$ prevents problems with nested occurrences of **run**$_f$. For example when a evaluates **run**$_f$ (**run**$_f$ e_1) to **run**$_f$ e_2, term e_2 might be contain true-casts for a. Hence V must be greater than a to ensure preservation.

Finally, **asbits** transforms an annotated ciphertext with **for** type into bare a ciphertext with type **bits**. This operator is typed as follows.

$$\frac{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash \diamond \qquad E \mathrel{|\!\sim} e : t\;\textsf{\textbf{for}}\;a}{\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash \textsf{\textbf{asbits}}\;e : \textsf{\textbf{bits}}}$$

The first premise maintains the invariant that the typing judgment's subjects are well-formed. The second premise uses a liberal over-approximation of typing to check that e is almost a t **for** a. The approximation, formalized in Figure 4.7, types variables and bare encryptions as usual, but always trusts the annotation on true or justified casts. It's sound to use the approximation here because **asbits** dynamically discards casts, returning the underlying ciphertexts; **asbits** launders bad **for**s into good **bits**. The typing rule is desirable because the typing of **asbits** e is independent of the facts context and statically available key, a useful property for defining mobile code.

It appears sound to allow any term to have type **bits**. Type **bits** would function like type Dynamic (Abadi et al., 1991). However, leaving this out of Aura$_{\text{conf}}$ provides two major benefits. First, Aura$_{\text{conf}}$'s type system is both syntax directed and assigns terms unique types—useful properties that would be be lost. Second, using type Dynamic requires a general typecase mechanism, and Aura$_{\text{conf}}$ has enough features already!

Basic metatheory and soundness

Aura$_{\text{conf}}$ satisfies two important properties: syntactic soundness and noninterference. Syntactic soundness guarantees that all well-typed programs have a well-defined evaluation semantics. Noninterference (Volpano et al., 1996; Askarov et al., 2008), states that a program's outputs are not affected (up to a natural equivalence induced by cryptography) by inputs intended to be secret. Aura$_{\text{conf}}$'s type system has several non-standard aspects; consequently, the technical statements and proofs of these properties are novel.

Except as noted, all properties of Aura$_{\text{conf}}$ are formalized as constructive proofs in the Coq proof assistant.

The following two properties are needed to prove soundness and, as discussed above, impact the design of Aura$_{\text{conf}}$'s type system.

Lemma 8 (Promotion)**.** *Suppose* $\Sigma; \mathcal{F}; W \,|\, E; V_1; U_1 \vdash e : t$*, and world inequalities* $U_1 \sqsubseteq U_2$ *and* $V_1 \sqsubseteq V_2$ *hold. Furthermore, assume* $\Sigma \vdash \diamond$ *and* $\Sigma; \mathcal{F}; W \,|\, E; V_2; U_2 \vdash \diamond$*. Then* $\Sigma; \mathcal{F}; W \,|\, E; V_2; U_2 \vdash e : t$*.*

Lemma 9 (Pure values)**.** *Suppose* $\Sigma \vdash \diamond$ *and* $\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash v : t$*. If val* v *then* $\Sigma; \mathcal{F}; W \,|\, E; V; \bot \vdash v : t$*.*

The substitution property, stated below, follows Jia and Walker (2003). Note that we require that v's statically available key matches the x's **at** annotation.

Lemma 10 (Substitution)**.** *Assume* $\Sigma \vdash \diamond$ *and* $\Sigma; \mathcal{F}; W \,|\, E, z : u\,\textbf{at}\,V_0; V; U \vdash e : t$*. If val* v *and* $\Sigma; \mathcal{F}; W \,|\, E; V_0; U \vdash v : u$*, then* $\Sigma; \mathcal{F}; W \,|\, E; \{v/z\}V; \{v/z\}U \vdash \{v/z\}e : \{v/z\}t$*.*

Stating preservation and progress requires defining when a term has reached an exceptional state. This is intended to occur only after a decryption failure, and identifies a proof to be used when diagnosing the failure. We write $e \text{ blames } p$ when (**fail** p) is a subterm of e, not located under a **return**$_f$ or a **bind**$_f$. In Coq this is defined as an inductive predicate over the syntax of terms. In principle this could also be handled with an exception that programs could catch for error recovery. The type system is designed so programs that have reached a blame-state are no longer typeable. This property simplifies the statement of progress.

Lemma 11. *If* $\Sigma; \mathcal{F}_0; W \,|\, E; V; U \vdash e : t$ *then there is no* p *such that* e blames p*.*

Preservation states that if a well-typed term steps either the resulting term has the same type, or else a decryption failed and the evaluator's state identifies a proof for blame assignment. Here notation $\mathcal{F}_0 +\!\!+ \mathcal{F}$ denotes a fact context containing the elements of $\mathcal{F}_0$ and $\mathcal{F}$.

Lemma 12 (Preservation)**.** *Assume* $\Sigma \vdash \diamond$ *and* $\Sigma; \mathcal{F}_0; W \,|\, E; V; U \vdash e : t$*. Then* $\Sigma; \mathcal{F}_0; W \vdash \{\!|e, n|\!\} \mapsto \{\!|e', n'|\!\}$ *learning* $\mathcal{F}$ *implies either* $\Sigma; \mathcal{F}_0 +\!\!+ \mathcal{F}; W \,|\, E; V; U \vdash e' : t$ *or there exists* p *such that* e' blames p*.*

The above lemma misses an important aspect of evaluation. Running an Aura$_{\text{conf}}$ program doesn't simply reduce an input term to a result; it also generates a sequence of new facts. There are terms that typecheck under bad fact contexts, but get stuck at evaluation. Thus we must ensure that newly generated facts are, in the following sense, semantically valid.

Definition 13 ($\text{valid}_\Sigma\, \mathcal{F}$)**.** *We write* $\text{valid}_\Sigma\, \mathcal{F}$ *when both the following hold. First,* $\Sigma \vdash \diamond$*. Second, for every* $\mathcal{E}(a, e, n) : t\ \textbf{for}\ b$ *in* $\mathcal{F}$ *it is the case that* $a = b$ *and* $\Sigma; \cdot; b \,|\, \cdot; b; b \vdash e : t$*.*

Intuitively this predicate holds when decrypting each ciphertext in a fact context would validate the declared types. Certain bogus facts, say `"hello"` : **int**, aren't harmful to soundness, and are ignored. The empty fact context is trivially valid.

Importantly $\text{valid}_\Sigma\, \mathcal{F}$ is not defined as a typing judgment because its truth, in general, may only be ascertained with access to every principal's private key. Such a property is useless when implementing a typechecker. Thus it is better to consider validity as a semantic property existing beside but distinct from Aura$_{\text{conf}}$'s type system.

The following lemma shows facts generated during reduction are valid.

Lemma 14 (New Fact Validity)**.** *Assume* $\Sigma \vdash \diamond$ *and* $\Sigma; \mathcal{F}_0; W \,|\, E; V; U \vdash e : t$*. Then* $\text{valid}_\Sigma\, \mathcal{F}_0$ *and* $\Sigma; \mathcal{F}_0; W \vdash \{\!|e, n|\!\} \mapsto \{\!|e', n'|\!\}$ *learning* $\mathcal{F}$ *implies* $\text{valid}_\Sigma\, \mathcal{F}$*.*

We assume that, as shown for Aura in Chapter 3, type checking Aura$_{\text{conf}}$ is decidable. This is a reasonable conjecture because Aura$_{\text{conf}}$ is syntax directed and designed with decidability in mind. Transliterating the earlier proof would be tedious but should yield no deep difficulties or insights. Decidability is of independent theoretic interest, but also matters because evaluating $\textbf{run}_f$

$$\boxed{W \vdash t_1 \simeq t_2}$$

$$\frac{}{W \vdash x \simeq x}\ \textsc{sim-Var} \qquad\qquad \frac{W \vdash t_{11} \simeq t_{12} \qquad W \vdash t_{21} \simeq t_{22}}{W \vdash (t_{11}\ t_{21}) \simeq (t_{12}\ t_{22})}\ \textsc{sim-App}$$

$$\frac{a \sqsubseteq W \qquad W \vdash e_1 \simeq e_2}{W \vdash \mathcal{E}(a, e_1, n_1) \simeq \mathcal{E}(a, e_2, n_2)}\ \textsc{sim-Decrypt}$$

$$\frac{a \not\sqsubseteq W \qquad b \not\sqsubseteq W}{W \vdash \mathcal{E}(a, e_1, n_1) \simeq \mathcal{E}(b, e_2, n_2)}\ \textsc{sim-Opaque}$$

Figure 4.8: Selected rules from the definition of similar terms

dynamically invokes the type checker. Were the type system not decidable, $\textbf{run}_f$ could instead conservatively approximate; otherwise the progress lemma would not hold.

Conjecture 15 (Decidability)**.** *If* $\Sigma \vdash \diamond$ *then it is decidable if* $\Sigma; \mathcal{F}; W \,|\, E; V; U \vdash e : t$.

The Aura$_{\text{conf}}$ statement of progress follows. Note that it describes the behavior of terms that are well-typed using a valid fact context. Additionally, any simple world greater than U and V—that is with the private keys specified by the soft decryption limit and effect label—has enough authority to step a program without getting stuck.

Lemma 16 (Progress)**.** *Assume Conjecture 15 holds. Assume also that* $\Sigma \vdash \diamond$, $\text{valid}_{\Sigma}\, \mathcal{F}_0$, *and* $\Sigma; \mathcal{F}_0; W_0 \,|\, E; V; U \vdash e : t$. *Now suppose* W *is a simple world where* $U \sqsubseteq W$ *and* $V \sqsubseteq W$. *Then either* e *is a value, or there exist* e', n', *and* $\mathcal{F}$ *where* $\Sigma; \mathcal{F}_0; W \vdash \{\!|e, n|\!\} \;\mapsto\; \{\!|e', n'|\!\}$ *learning* $\mathcal{F}$.

Lemmas 12, 14, and 16 together imply Aura$_{\text{conf}}$ is sound.

Noninterference

Noninterference properties, which state that a program's secret inputs do not influence its public outputs, are a common way of defining security for programming languages (Volpano et al., 1996; Vaughan and Zdancewic, 2007). Such properties are formalized by saying programs which differ only in their secret components are *similar* and showing that similar terms reduce to similar values. The following develops a noninterference property for Aura$_{\text{conf}}$.

Aura$_{\text{conf}}$ similarity is defined relative to particular set of keys used to analyze ciphertexts. Figure 4.8 gives the key rules from the definition of similarity. Most often, two terms are related when they are identical, as in SIM-VAR, or share a top-level constructor with similar subterms, as in SIM-APP. The figure elides a tedious quantity of rules implementing this scheme. Similarity is more interesting for ciphertexts. Rule SIM-DECRYPT finds two ciphertexts similar when they are encrypted with the same key, can be decrypted by W (captured by premise $a \sqsubseteq W$), and have similar payloads. This formalizes the idea that encrypting similar terms should yield similar results. Finally, SIM-OPAQUE states two ciphertexts are similar when neither can be decrypted. This captures the intuition that ciphertexts are black boxes, immune to analysis without a key. We implicitly assume ciphertexts are (randomly) padded such that ciphertext length cannot be used at a side channel. A faithful implementation will require care to properly handle Aura$_{\text{conf}}$'s rich data structures.

The following lemma gives Aura$_{\text{conf}}$'s noninterference property. It considers running two terms, e_1 and e_2, that step without error under authority W. If the terms are similar at W (or any higher world W_0), the resulting terms, e'_1 and e'_2, are similar as well. In particular, running a program twice with two different confidential inputs yields outputs that only be distinguished with a sufficiently privileged private key.

Lemma 17 (Noninterference)**.** *Assume* $\Sigma \vdash \diamond$ *and* $\Sigma; \mathcal{F}_1; W \mid \cdot; V; U \vdash e_1 : k_1$*. Pick* W_0 *and* e_2 *where* $W_0 \vdash e_1 \simeq e_2$ *and* $W \sqsubseteq W_0$*. If*

- $\Sigma; W; \mathcal{F}_1 \vdash \{\!|e_1, n_1|\!\} \mapsto \{\!|e'_1, n'_1|\!\}$ *learning* $\mathcal{F}'_1$*,*
- $\Sigma; W; \mathcal{F}_2 \vdash \{\!|e_2, n_2|\!\} \mapsto \{\!|e'_2, n'_2|\!\}$ *learning* $\mathcal{F}'_2$*,*
- *there is no* p *such that* e'_1 blames p*, and*
- *there is no* p *such that* e'_2 blames p*,*

then $W_0 \vdash e'_1 \simeq e'_2$*.*

4.5 Discussion

Information-flow and Aura

Information-flow analyses (Sabelfeld and Myers, 2003) inspired this chapter's goal of augmenting Aura to handle confidential data. However, while these techniques influenced and informed the design of Aura$_{\text{conf}}$, they cannot be directly applied.

In standard information-flow systems, programmers use *labels* to express confidentiality and integrity constraints on data, and the language's typing judgment is specialized to deal with these labels (Volpano et al., 1996). Well-typed terms are correct by construction; they satisfy noninterference. (However, increasingly expressive information-flow languages often satisfy variously weakened versions of the property (Chong and Myers, 2006; Myers et al., 2006).) Most conventional information-flow languages are limited by a focus on closed systems: the programmer must, for example, manually encrypt confidential data leaving the program with an unsafe *declassification* operator. Aura as described in Chapter 3 can encode this style of of information flow analysis (Jia and Zdancewic, 2009).

In previous work with Steve Zdancewic (2007), I described an information-flow language, SImp, suitable for programming in open systems. SImp resolves the mismatch between policy specification and enforcement by connecting information flow labels directly with public key cryptography. Policies and data may be combined into *packages* that use digital signatures and encryption to ensure only principals with appropriate keys may access data.

SImp policies are specified by annotating data values and heap locations with semantically rich *labels*. Labels are lists of security sublabels with owner, confidentiality, and integrity components. Sublabel $o : \overline{r} \,!\, \overline{w}$ means owner o certifies that any principal in set $\overline{r}$ may read from the associated location, and any principal in set $\overline{w}$ may write. Full labels allow (groups of) principals to read or write when each sublabel is satisfied. This is a variant of Myers and Liskov's (2000) decentralized label model (DLM).

Although SImp's design influenced Aura$_{\text{conf}}$, its technical mechanisms could not be adopted wholesale. Information-flow analysis with DLM labels, the basis for SImp, provides a very different model of declarative information security than Aura. In particular Aura's **says** monad decorates propositions to express *endorsement*, while SImp's integrity sublabels described tainted data. While intuitively related, these concepts demand different treatments. It is unclear how to understand DLM owners in Aura. Additionally, interpreting the semantics of a DLM label—that is, calculating its effective reader and writer sets—requires knowledge of the global delegation relation, or "acts-for hierarchy," information that cannot be reliably obtained in Aura's distributed setting. SImp's design did provide direct inspiration for several aspects of Aura$_{\text{conf}}$, including declarative policy specification, a key-based notion of identity, and automatic encryption.

On Noninterference

Aura$_{\text{conf}}$'s noninterference property (Lemma 17) is weak in the following sense. It discusses what happens when a pair of terms with different secrets successfully take a step, but does not deal with the situation in which one steps successfully and the other fails. The reason is subtle. Consider the following terms:

$$\begin{array}{rcl} \mathsf{ok} & : & \mathcal{E}(\mathsf{a},\ \text{"hi"},\ 1) \rightarrow \textbf{Prop} \\ e_1 & \equiv & \mathsf{ok}(\ \mathcal{E}(\mathsf{a},\ \text{"hi"},\ 1)) \\ e_2 & \equiv & \mathsf{ok}(\ \mathcal{E}(\mathsf{a},\ \text{"hi"},\ 2)) \end{array}$$

Terms e_1 and e_2 represent differently randomized encryptions of the same string. It's intuitively appealing that these are similar for purposes of noninterference, and indeed $\mathsf{a} \vdash e_1 \simeq e_2$. However term e_1 is well-typed, but e_2 is not. Terms like these can cause $\textbf{run}_f$ to show different failure behavior when applied to similar terms. Consequently, Lemma 17's definition of noninterference is an example of termination-insensitive noninterference (Askarov et al., 2008).

Termination insensitivity is required because Aura$_{\text{conf}}$ and its metatheory have the following three properties. First, the language can express singleton types on ciphertexts—useful in general and necessary for **isa** propositions. Second, it features a typesafe decryption operator that works at arbitrary types—a design goal. Third, the similarity relation is aligned with standard Dolev-Yao (1983) cryptanalysis. While it's possible to alter one of these properties to induce a stronger form of noninterference, such a change appears counterproductive.

4.6 Related Work

Modal type theory

Modal logics provide a framework to describe the way in which a proposition holds. Common modalities can specify that a sentence is necessarily vs. possibly true or that a condition will be met eventually vs. from-now-on. In the vernacular of Kripke structures this is a technique for reasoning about different worlds, a terminology that Aura$_{\text{conf}}$ borrows (Goldblatt, 2003). Pfenning and Davies (2001) introduced a constructive, type-theoretic treatment of modal logic. Their account focuses on the logical foundations of the system. Jia and Walker (2004) studied a similar theory from a distributed-programming perspective, interpreting modal operators as specifying the locations at which code may run. While Pfenning discusses three judgments, truth, validity and possibility, Jia presents an indexed judgment form that can describe a large quantity of locations. Murphy's (2008) dissertation describes a full-scale programming language based on these ideas.

The systems above have an absolute static semantics. That is, although executing code may depend on location or resource availability, checking that a type (or proposition) is well formed can happen anywhere. Aura$_{\text{conf}}$'s ability to make typing more precise using statically available keys appears novel.

Cryptography and programming languages

One intended semantics for Aura$_{\text{conf}}$ implements objects of form $\textbf{sign}(\mathsf{a},\ \mathsf{P})$ as digital signatures and objects like $\mathcal{E}(\mathsf{a},\ \mathsf{e},\ \mathsf{n})$ as ciphertext. All cryptography occurs at a lower level of abstraction than the language definition. This approach has previously been used to implement declarative information flow policies (Vaughan and Zdancewic, 2007). An alternative approach is to treat keys as types or first class objects and to provide encryption or signing primitives in the language (Askarov et al., 2006; Chothia et al., 2003; Smith and Alpízar, 2006; Laud and Vene, 2005; Laud, 2008; Fournet and Rezk, 2008). Such approaches typically provide the programmer with additional flexibility but complicate the programming model.

Askarov, Hedin, and Sabelfeld (Askarov et al., 2006) recently investigated a type system for programs with encryption and with the property that all well typed programs are noninterfering. Their work differs from ours in several ways. They treat encryption, decryption, and key generation as language primitives. In contrast, we use cryptography implicitly to implement high-level language features. Askarov's language appears superior for modeling cryptographic protocols, and ours provides a cleaner and simpler interface for applications programming.

Chothia, Duggan, and Vitek (Chothia et al., 2003) examine a combination of DLM-style policies and cryptography, called the Key-Based DLM (KDLM). Their system, like Askarov's, provides an extensive set of language level cryptographic primitives and types inhabited by keys. Similarly to Aura$_{\text{conf}}$, KDLM security typing is nominal—labels have names and each name corresponds to a unique cryptographic key. While they prove type soundness, Chothia and colleagues do not provide more specific security theorems such as noninterference.

Sumii and Pierce (2004) studied λ_{seal}, an extension to lambda calculus with terms of form $\{e\}_{e'}$, meaning e sealed-by e', and a corresponding elimination form. Like Askarov and colleagues, they make seal (i.e. key) generation explicit in program text; however their dynamic semantics, which include runtime checking of seals, is simpler than Askarov's. Additionally, λ_{seal} includes black-box functions that analyze sealed values, but cannot be disassembled to reveal the seal (key). It is unclear how to implement these functions using cryptography.

Heintze and Riecke's (1998) SLam calculus is an information flow lambda calculus in which the right to read a closure corresponds to the right to apply it. This sidesteps the black-box function issue from λ_{seal}. In SLam, some expressions are marked with the authority of the function writer. The annotations control declassification, and, we conjecture, are analogous to the pretranslated labels in the SImp language (Section 4.5). Additionally SLam types have a nested form where, for example, the elements in a list and the list itself may be given different security annotations. Combined with **pack**, such nesting could facilitate defining data structures with dynamic and heterogeneous security properties.

We use the algebraic Dolev-Yao model to study the connection connection between information flow and cryptography. Laud and Vene (2005) examined this problem using a computational model of encryption. More recently, Smith and Alpízar (2006) extended this work to include a model of decryption. They prove noninterference for a simple language without declassification (or packing) and a two-point security lattice. Like Chothia and colleagues, they map labels to fixed keys.

Abadi and Rogaway (2002) proved that Dolev-Yao analysis is sound with respect to computational cryptographic analysis in a setting similar to Vaughan and Zdancewic's (2007). However, there are several significant differences between these approaches. In particular, Abadi and Rogaway do not discuss public key cryptography, which we use extensively. Backes and Pfitzmann (2005) with Waidner (Backes et al., 2003) have also investigated the connection between symbolic and computational models of encryption. They define a Dolev-Yao style library and show that protocols proved secure with respect to library semantics are also secure with respect to computational cryptographic analysis. Likewise Barthe et al. (2009) have published a Coq formalization of several cryptographic algorithms, including ElGamal digital signatures. These techniques and artifacts might provide an excellent foundation for further rigorous analysis of $\text{Aura}_{\text{conf}}$.

Chapter 5

Conclusion

Aura$_0$, Aura, and Aura$_{\text{conf}}$ were designed to explore the technical essence of integrating access control, audit, and confidentiality mechanisms in a programming language. These languages also provide a concrete platform for further experiments in declarative information security. This chapter first looks back at these languages, then forward to possible extensions.

5.1 Summary

This dissertation presented the Aura family of programming languages, which use a combination of dependent types and cryptography to support access control, confidentiality and audit. Programs written in these languages are capable of specifying security properties that are automatically enforced by language semantics, freeing the programmer from, for example, explicit key management. Policies are stated using the propositional fragment of Aura's type system. This provides a foundational and expressive language for policy definition.

Aura's audit model and the formalism given in Chapter 2 represent a first step toward the development of a precise theory of audit for secure systems. Understanding this heretofore neglected topic may enable more rigorous analysis of security sensitive programs.

Aura$_{\text{conf}}$'s treatment of cryptography includes several novel elements. Because Aura$_{\text{conf}}$ uses statically available keys and fact contexts to augment compile-time typechecking, it places unusual demands on its type system. In particular, the very notion of well-typedness is dependent on which keys are available statically, and it is challenging to predict where a term can typecheck. Additionally, evaluation can also use private keys, and programs will get stuck if run in the wrong context. Aura$_{\text{conf}}$ answers "where can a term be typechecked?" and "where can it be run?" by combining, in a new way, ideas from modal type theory and type-and-effect analysis.

5.2 Possible Extensions

Programming with signature revocation and expiry

In Aura signatures are permanent—once created they are valid indefinitely and unconditionally. This is undesirable when a proposition is erroneously signed, a private key is compromised, or an institutional policy changes. Implemented digital signature protocols, including OpenPGP (Callas et al., 2007), address this in two ways: by allowing signatures to be annotated with an expiration time, and by defining revocation certificates which witness that a particular signature is invalid. These concepts could be applied in Aura to enforce time-limited delegation or to implement mutable policies based on revocation.

Elegantly integrating signature expiration and revocation into Aura is a design challenge. The simplest implementation strategy is to maintain the logic's current form and to accept that some proofs will expire unexpectedly at runtime. This approach requires that operations dynamically validate signatures before logging, thus making all kernel operations partial. The situation could be improved with a transaction mechanism that can ensure proofs are (and will be) current within some lexical scope.

Alternatively, signature revocation could be modeled with linear or affine types, see Bauer et al. (2006) for an authorization logic with linearity constraints. Linear and affine objects must be consumed either exactly or at most once, and are appropriate for granting access to a resource a fixed number of times.

DeYoung, Garg, and Pfenning (2008) studied a constructive access-control logic with explicit time intervals. Their syntax includes propositions of the form $P@[I]$, meaning "P holds during interval I." The logic requires that intervals are consistent with only weak algebraic axioms, and the approach generalizes to constraints unrelated to time. Incorporating these ideas could allow Aura to address temporal policies while providing a robust platform for future extensions to its authorization logic.

Computation language enhancements

The design of both core Aura and Aura$_{\text{conf}}$ could be enhanced in several ways. The following describes a selection of interesting directions.

Aura is a verbose language, with many type annotations and explicit type instantiations. While this is desirable for an intermediate language, it suggests several challenges that must be met in providing surface syntax. Because Aura generalizes System F, type inference is likely undecidable. (Recall that Chapter 3 provided decidability of type *checking*.) However, an incomplete inference algorithm guided by annotations, in the style of Pierce and Turner (1998), would be an essential component of a surface language. Aura's proof-passing style also suggests investigating heuristics for proof inference. Aura$_{\text{conf}}$'s type system poses an additional challenge, in that surface language would require a user-friendly way to identify and report type errors due to problems with statically available keys, soft decryption limits, and effect labels.

Aura's restrictions on dependency and its weak notion of type refinement facilitate decidable typechecking. However, it's likely that more liberal variants of the type system would retain decidability as well. Future versions of Aura could be extended to include a generalized **match** construct, perhaps inspired by GADTs (Peyton Jones et al., 2006) or Agda (Norell, 2007). Similarly, Aura's conservative positivity constraint on inductive proposition definitions could be relaxed following Paulin-Mohring (1993). This would allow propositions to represent more sophisticated predicates. However even with a more expressive propositional fragment, Aura would not be tuned for general-purpose mathematical reasoning.

Currently Aura$_{\text{conf}}$ provides a fail-stop semantics: decryption failures lead to an uncatchable, fatal exception. It would be better to handle these errors programmatically, and a variety of techniques could be brought to bear. Most promising, Aura$_{\text{conf}}$ could be extended with a general purpose exception mechanism like ML's or Java's. Doing this properly requires merging exceptions with effects analysis with higher-order types, a design space that is still being explored (Leroy and Pessaux, 2000; Blume et al., 2008; Benton and Buchlovsky, 2007). A more readily implementable scheme might have $\textbf{run}_f$ return a discriminated union.

Aura$_{\text{conf}}$'s worlds provide a simple model of key management. Programs may be run and typechecked using zero or one statically available keys. The typing judgment additionally allows $\top$ to represent “all keys” in effect labels and soft decryption limits. However worlds are treated primarily as lattice elements and it appears interesting to define worlds using a richer structure, such as sets of keys or DLM-inspired labels. This generalization would require making (hopefully) straightforward modifications to the language metatheory and, more interestingly, carefully designing an expressive and practical security lattice. Were Aura$_{\text{conf}}$ re-engineered to use a more expressive formulation of worlds, it might also be fruitful to index the **for** and **says** monads by worlds instead of principals.

References

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.

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.

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.

Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, April 1991. Also appeared as SRC Research Report 47.

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.

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.

Apache Tutorial: .htaccess files. The Apache Software Foundation, 2009. Available from `http://httpd.apache.org/docs/2.2/howto/htaccess.html`.

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. ISBN 1-58113-148-8.

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

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88312-8. doi: http://dx.doi.org/10.1007/978-3-540-88313-5_22.

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

S. Axelsson, U. Lindqvist, U. Gustafson, and E. Jonsson. An approach to UNIX security logging. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 62–75, 1998. URL `citeseer.ist.psu.edu/axelsson98approach.html`.

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.

Michael Backes and Birgit Pfitzmann. Relating symbolic and cryptographic secrecy. *IEEE Trans. Dependable Secur. Comput.*, 2(2):109–123, 2005. ISSN 1545-5971. doi: http://dx.doi.org/10.1109/TDSC.2005.25.

Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 220–230, Washington D.C., USA, 2003. ACM Press. ISBN 1-58113-738-9. doi: http://doi.acm.org/10.1145/948109.948140.

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.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 90–101, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480894.

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

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 2005a.

Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security & Privacy*, pages 81–95, May 2005b. URL `http://www.ece.cmu.edu/~lbauer/papers/2005/sp2005-distributed-proving.pdf`.

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

Lujo Bauer, Lorrie Faith Cranor, Robert W. Reeder, Michael K. Reiter, and Kami Vaniea. A user study of policy creation in a flexible access-control system. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 543–552, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-011-1. doi: http://doi.acm.org/10.1145/1357054.1357143.

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

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 17–32, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3182-3. doi: http://dx.doi.org/10.1109/CSF.2008.27.

Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '07)*, Nice, France, January 2007. ACM.

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

Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet programming: security issues for mobile and distributed objects*, pages 185–210. Springer-Verlag, London, UK, 1999a. ISBN 3-540-66130-1.

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, 1999b.

Matthias Blume, Umut A. Acar, and Wonseok Chae. Exception handlers as extensible cases. In *Proceedings of the Sixth ASIAN Symposium on Programming Languages and Systems (APLAS 2008)*, Bangalore, India, 2008. To appear.

J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007. URL `http://www.ietf.org/rfc/rfc4880.txt`.

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.

Steve Chong and Andrew C. Myers. Decentralized robustness. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 242–253, Los Alamitos, CA, USA, July 2006.

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.

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.

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

Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, 2008. Free online edition, version 1.5. Avaliable from `http://svnbook.red-bean.com/`.

D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. URL `http://www.ietf.org/rfc/rfc5280.txt`.

The Coq Proof Assistant Reference Manual. The Coq Development Team, LogiCal Project, 2006.

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

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

N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34 (5):381–392, 1972. Also published in the Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam, series A, 75, No. 5.

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

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.

D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

Derek Dreyer and Andreas Rossberg. Mixin' up the ml module system. *SIGPLAN Not.*, 43(9): 307–320, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1411203.1411248.

C. Ellison and B. Schneier. Ten risks of pki: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.

C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. RFC 2693 (Proposed Standard), 1999. URL `http://www.ietf.org/rfc/rfc2963.txt`.

Sandro Etalle and William H. Winsborough. A posteriori compliance control. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 11–20, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-745-2. doi: http://doi.acm.org/10.1145/1266840.1266843.

Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *Proc. 35rd ACM Symp. on Principles of Programming Languages (POPL)*, pages 323–335, New York, NY, USA, 2008. ACM.

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.

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.

Deepak Garg and Frank Pfenning. A proof-carrying file system. Technical Report CMU-CS-09-123, Computer Science Department, Carnegie Mellon University, June 2009.

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

Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-506-2. doi: http://doi.acm.org/10.1145/1536414.1536440.

Herman Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In *TYPES '94: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 14–38, London, UK, 1995. Springer-Verlag. ISBN 3-540-60579-7.

Robert Goldblatt. Mathematical modal logic: a view of its evolution. *J. of Applied Logic*, 1(5-6): 309–392, 2003. ISSN 1570-8683. doi: http://dx.doi.org/10.1016/S1570-8683(03)00008-9.

Nataliya Guts, Cédric Fournet, and Francesco Zappa Nardelli. Reliable evidence: Auditability by typing. In *ESORICS 2009: 14th European Symposium on Research in Computer Security*, number 5789 in Lecture Notes in Computer Science, pages 168–183. Springer-Verlag, 2009.

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. ISBN 3-540-67895-6.

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

Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-979-3. doi: http://doi.acm.org/10.1145/268946.268976.

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.

Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322217.322230.

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

Limin Jia and David Walker. Modal proofs as distributed programs. Technical Report TR-671-03, Princeton University, August 2003.

Limin Jia and David Walker. Modal proofs as distributed programs (extended abstract). In *Programming Languages and Systems: 13th European Symposium on Programming (ESOP 2004)*, 2004.

Limin Jia and Steve Zdancewic. Encoding information flow in aura. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 17–29, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-645-8. doi: http://doi.acm.org/10.1145/1554339.1554344.

Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: A programming language for authorization and audit. In *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 27–38, 2008.

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

Peeter Laud. On the computational soundness of cryptographically masked flows. *SIGPLAN Not.*, 43(1):337–348, 2008. ISSN 0362-1340.

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.

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. ISBN 1-59593-575-4.

Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/349214.349230.

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

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

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: http://doi.acm.org/10.1145/73560.73564.

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

Tom Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon, January 2008. URL `http://tom7.org/papers/`. (draft).

Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*. IEEE Press, July 2004.

Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 2006. To appear.

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.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3):1–49, 2008. ISSN 1529-3785. doi: http://doi.acm.org/10.1145/1352582.1352591.

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.

Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, London, UK, 1993. Springer-Verlag. ISBN 3-540-56517-5.

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

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. ISBN 1-59593-309-3.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical. Structures in Comp. Sci.*, 11(4):511–540, 2001. ISSN 0960-1295. doi: http://dx.doi.org/10.1017/S0960129501003322.

Benjamin C. Pierce and David N. Turner. Local type inference. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California*, 1998.

Dörte K. Rappe. *Homomorphic Cryptosystems and Their Applications*. PhD thesis, University of Dortmund, Germany, 2004.

Andreas Rossberg. *Typed Open Programming – A higher-order, typed approach to dynamic modularity and distribution*. Phd thesis, Universität des Saarlandes, Saarbrücken, Germany, January 2007. Preliminary version.

Andreas Rossberg, Guido Tack, and Leif Kornstaedt. Status report: HOT pickles, and how to serve them. In Claudio Russo and Derek Dreyer, editors, *2007 ACM SIGPLAN Workshop on ML*, pages 25–36. ACM, 2007.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

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

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

Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. *J. Funct. Program.*, 17(4-5):547–612, 2007. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796807006442.

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, Alexandria, Virgina, USA, November 2006.

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. ISBN 1-59593-393-X.

Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *Principals of Programming Languages*, Venice, Italy, January 2004.

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.

Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F♯*. Apress, Berkeley, CA, 2007.

J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Conference on Logic in Computer Science (LICS'92)*. IEEE Press, June 1992.

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

Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Steve Zdancewic. Evidence-based audit. In *Proc. of the 21th IEEE Computer Security Foundations Symposium*, pages 177–191, 2008. Extended version available as U. Pennsylvania Technical Report MS-CIS-08-09.

Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

Geoffrey Washburn. Cause and effect: type systems for effects and dependencies. Technical Report MS-CIS-05-05, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2005.

Geoffrey Alan Washburn. *Principia narcissus: how to avoid being caught by your reflection*. PhD thesis, Philadelphia, PA, USA, 2007.

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

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

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.

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

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.

Tatu Ylonen. SSH – secure login connections over the Internet. In *The Sixth USENIX Security Symposium Proceedings*, pages 37–42, San Jose, California, 1996.

Appendix A

Proofs for Aura$_0$

This section gives proofs of subject reduction and strong normalization for Aura$_0$ propositions.

Subject reduction for Aura$_0$ propositions

Lemma 18 (Weakening). *If* $\Sigma; \Gamma, \Gamma' \vdash t_1 : t_2$ *and* $\Sigma \vdash \Gamma, x : t_3, \Gamma'$ *then* $\Sigma; \Gamma, x : t_3, \Gamma' \vdash t_1 : t_2$.

Proof. By structural induction on the typing derivation. □

Lemma 19 (Inversion Var—Same). *If* $\Sigma; \Gamma, z : u, \Gamma' \vdash z : s$ *then* $u = s$.

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

Lemma 20 (Inversion Var—Different). *If* $\Sigma; \Gamma, z : u, \Gamma' \vdash x : s$ *and* $x \neq z$ *then* $x : s \in \Gamma$ *or* $x : s \in \Gamma'$.

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

Lemma 21 (Variables closed in context). *If* $\Sigma; \Gamma \vdash s : t$ *and* $x \in \mathit{fv}(t) \cup \mathit{fv}(s)$ *then* $x \in \mathit{dom}(\Gamma)$.

Proof. Proof by induction on the typing derivation. □

Lemma 22 (Context Ordering). *If* $\Sigma \vdash \Gamma, z : u$ *and* $x : s \in \Gamma$ *then* $z \notin \mathit{fv}(s)$.

Proof. By the definition of $\in$, we know $\Gamma = \Gamma_1, x : s, \Gamma_2$. The well-formedness of $\Gamma_1, x : s, \Gamma_2, z : u$ shows that (1) $dom(z) \notin \Gamma_1$ and (2) $\Sigma; \Gamma_1 \vdash s : k$ for some k. From these and Lemma 21 we can conclude $z \notin fv(s)$. □

Lemma 23 (Well-formedness)**.** *(1) If* $\Sigma; \Gamma \vdash t : s$ *then* $s \in \{\mathsf{Kind}^{\boldsymbol{P}}, \mathsf{Kind}^{\boldsymbol{T}}\}$ *or there exists* k *such that* $\Sigma; \Gamma \vdash s : k$*.*

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

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

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

Lemma 24 (Substitution, strong form for induction)**.** *Assume* $\Sigma; \Gamma \vdash t_u : u$*. Then*

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

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

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

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

Case T-VAR. Suppose $t = z$. Then, by Lemma 19 $s = u$. Therefore it suffices to show $\Sigma; \Gamma, \{t_u/z\}\Gamma' \vdash t_u : u$, which we get by applying weakening (finitely many times) to the assumption $\Sigma; \Gamma \vdash t_u : u$. Instead suppose $t = x \neq z$. Then we must show $\Sigma; \Gamma, \{t_u/z\}\Gamma' \vdash x : \{t_u/z\}s$. By Lemma 20, either $x : s \in \Gamma$ or $x : s \in \Gamma'$. Suppose $x : s \in \Gamma$. Then by Lemma 22 we find $z \notin fv(s)$ so $s = \{t_u/z\}s$. Thus it suffices to show $\Sigma; \Gamma, \{t_u/z\}\Gamma' \vdash x : s$, which follows from T-VAR, the induction hypothesis and the form of Γ. Lastly, consider the case that $x : s \in \Gamma'$. Then $x : \{t_u/z\}s \in \{t_u/z\}\Gamma'$, and we conclude using this, T-VAR, and the induction hypothesis.

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

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

Case T-SIGN. We have $t = \textsf{sign}(t_1, t_2)$ and $s = t_1 \textsf{ says } t_2$. From Lemma 21, we find $fv(t_1) = fv(t_2) = \emptyset$. Hence $\{t_u/z\}t = t$ and $\{t_u/z\}s = s$, so to use T-SIGN, we need only show $\Sigma \vdash \Gamma, \{t_u/z\}\Gamma'$. This follows immediately from the induction hypothesis.

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

The remaining cases are similar to T-PROP (T-TYPE, T-STRING, T-CONST, T-PRIN, T-LITSTR, T-LITPRIN), or T-LAM (T-ARR, T-PAIRTYPE), or are trivial (T-SAYS, T-RETURN, T-APP). □

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

Lemma 25 (Strengthening)**.** *If* $\Sigma; \Gamma, z : u, \Gamma' \vdash t : s$ *and* $\Sigma \vdash \Gamma, \Gamma'$ *and* $z \notin fv(t) \cup fv(s)$ *then* $\Sigma; \Gamma, \Gamma' \vdash t : s$.

Proof. Proof by structural induction on the typing relation.

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

Case T-PAIR. We have $s = \{x{:}s_1; s_2\}$ for some x, s_1, and s_2. By 23 and a simple case analysis, $\Sigma; \Gamma \vdash s : k$ where $k \in \{\textsf{Prop}, \textsf{Type}, \textsf{Kind}^{\textsf{P}}, \textsf{Kind}^{\textsf{T}}\}$. Thus $z \notin fv(k)$. Thus the case follows from the induction hypothesis and T-PAIR.

All other cases follow directly from the induction hypothesis. □

Lemma 26 (Subject Reduction)**.** *If* $\vdash t \rightarrow t'$ *and* $\Sigma; \Gamma \vdash t : s$ *then* $\Sigma; \Gamma \vdash t' : s$.

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

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

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

for some s_2 and s_3. So $s = \{t_3/x\}t_2$. That $\Sigma;\Gamma \vdash t' : s$ holds follows directly from Lemma 24 and the judgments written in the above derivation.

Case R-BINDS. We have $t = \textsf{\textbf{bind}}\ x\ =\ t_1\ \textsf{\textbf{in}}\ t_2$ and $t' = t_2$. Term t could only be typed by T-BIND, and inverting this rule gives $s = a\ \textsf{\textbf{says}}\ s_2$ and $\Sigma;\Gamma, x : s_1 \vdash t_2 : a\ \textsf{\textbf{says}}\ s_2$. Before concluding with Lemma 25, we must show $x \notin a\ \textsf{\textbf{says}}\ s_2$. This is a consequence of Lemma 21, and the hypothesis that $a\ \textsf{\textbf{says}}\ s_2$ is a type assignment in Γ.

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

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

By Lemma 24, we find $\Sigma;\Gamma \vdash \{t_1/x\}t_2 : \{t_1/x\}(t_0\ \textsf{\textbf{says}}\ s_2)$. The contrapositive of Lemma 21 shows $x \notin t_0\ \textsf{\textbf{says}}\ s_2$, so we can rewrite the above to $\Sigma;\Gamma \vdash t' : s$.

Case R-BINDC. We have $t = \textsf{\textbf{bind}}\ x\ =\ (\textsf{\textbf{bind}}\ y\ =\ t_1\ \textsf{\textbf{in}}\ t_2)\ \textsf{\textbf{in}}\ t_3$ and $s = u\ \textsf{\textbf{says}}\ s_3$. Following the Barendregt variable convention, assume $y \notin \mathit{fv}(\Gamma) \cup \{x\}$. Inverting the typing derivation twice shows, for some s_1 and s_2, that $\Sigma;\Gamma \vdash t_1 : u\ \textsf{\textbf{says}}\ s_1$, $\Sigma;\Gamma, y : s_1 \vdash t_2 : u\ \textsf{\textbf{says}}\ s_2$, $\Sigma;\Gamma, x : s_2 \vdash t_3 : u\ \textsf{\textbf{says}}\ s_3$, and $x \notin \mathit{fv}(s_3)$. With Lemma 18 we find $\Sigma;\Gamma, y : s_1, x : s_2 \vdash t_3 : u\ \textsf{\textbf{says}}\ s_3$. With Lemma 21, $y \notin \mathit{fv}(s_3)$. We conclude using T-BINDC twice.

The remaining cases follow directly from the induction hypothesis. □

Strong normalization for Aura$_0$ propositions

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

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

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

Definitions

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

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

The translation of Aura$_0$ terms to CC terms is defined in Figure A. The translation collapses **Kind$^{\mathbf{P}}$** and **Kind$^{\mathbf{T}}$** to the kind $\Box$ in CC, and **Prop**, **Type** to $*$. We translate all base types to **unit**, and constants to ( ). The interesting cases are the translation of DCC terms. The translation drops the monads, and translates the **bind** expression to lambda application. The term **sign**(t_1, t_2) has type t_1 **says** t_2; therefore, it has to be translated to a term whose type is the translation of t_2. One way to find such a term is to generate a fresh variable and assign its type to be the translation of t_2. The context Δ is used to keep track of the type mapping of those fresh variables generated. There are two cases in translation **sign**(t_1, t_2). In the first case, the variable we need has already been generated. In the second case, we need to generate a fresh variable and append its type binding to Δ_1 as the output context. To make proofs easier, we assume that the fresh variables are denoted by a *fvar*, not to be confused with the variable x.

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

Lemma 27 (Translation Weakening)**.** *If* $[\![t]\!]_{\Delta_1} = (s, \Delta_2)$*, unique*$(\Delta)$*, and* $(\Delta_1, \Delta_2) \subseteq \Delta$*, then* $[\![t]\!]_\Delta = (s, \cdot)$.

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

case: $t = \textsf{\textbf{sign}}(t_1, t_2)$.

By assumptions,

$$\textit{unique}(\Delta) \tag{1}$$

$$[\![\textsf{\textbf{sign}}(t_1, t_2)]\!]_{\Delta_1} = (x, \Delta_2)$$

$$\text{and } [\![t_2]\!]_{\Delta_1} = (s, \Delta_2),\ \ (\Delta_1, \Delta_2)(\textit{fvar}) = s \tag{2}$$

$$(\Delta_1, \Delta_2) \subseteq \Delta \tag{3}$$

By I.H. on t_2,

$$[\![t_2]\!]_\Delta = (s, \cdot) \tag{4}$$

By (2), (1), (3),

$$\Delta(\textit{fvar}) = s \tag{5}$$

By the rules for translation,

$$[\![\textsf{\textbf{sign}}(t_1, t_2)]\!]_\Delta = (\textit{fvar}, \cdot) \tag{6}$$

case: $t = \textsf{\textbf{sign}}(t_1, t_2)$.

By assumptions,

$$\textit{unique}(\Delta) \tag{1}$$

$$[\![\textsf{\textbf{sign}}(t_1, t_2)]\!]_{\Delta_1} = (\textit{fvar}_1, (\Delta_2, \textit{fvar}_1 : s))$$

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

$$\frac{\text{if } t \in \{\textbf{Kind}^{\mathbf{P}}, \textbf{Kind}^{\mathbf{T}}\}}{[\![t]\!]_\Delta = (\square, \cdot)} \qquad \frac{\text{if } t \in \{\textbf{Prop}, \textbf{Type}\}}{[\![t]\!]_\Delta = (*, \cdot)} \qquad \frac{\text{if } t \in \{\texttt{"a"}, \ldots, \texttt{A}\ldots\}}{[\![t]\!]_\Delta = ((\,), \cdot)}$$

$$\frac{\text{if } t \in \{\textbf{string}, \textbf{prin}\}}{[\![t]\!]_\Delta = (\textbf{unit}, \cdot)} \qquad [\![a]\!]_\Delta = (a, \cdot) \qquad [\![x]\!]_\Delta = (x, \cdot) \qquad \frac{}{[\![t_1 \;\textbf{says}\; t_2]\!]_\Delta = [\![t_2]\!]_\Delta}$$

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

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

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

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

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

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

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

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

Figure A.1: Translation of Aura$_0$'s terms to CC

and $[\![t_2]\!]_{\Delta_1} = (s, \Delta_2)$,

$$\nexists \mathit{fvar} \in dom(\Delta_2) \text{ s.t. } (\Delta, \Delta_2)(\mathit{fvar}) = s \tag{2}$$

$$(\Delta_1, \Delta_2, \mathit{fvar} : s) \subseteq \Delta \tag{3}$$

By I.H. on t_2,

$$[\![t_2]\!]_\Delta = (s, \cdot) \tag{4}$$

$$\boxed{[\![\Sigma]\!] = \Sigma'}$$

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

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

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

$$\frac{[\![\Gamma]\!]_\Sigma = (\Gamma', \Delta_1) \qquad \mathit{wf}(\Sigma, \Delta_1, \Delta_2) \qquad \mathit{unique}(\Sigma, \Delta_1, \Delta_2) \qquad [\![t]\!]_{\Sigma,\Delta_1,\Delta_2} = (s, \Delta_3)}{[\![\Gamma, v : t]\!]_\Sigma = ((\Gamma', v : s), (\Delta_1, \Delta_2, \Delta_3))}$$

Figure A.2: Translation of Aura$_0$ contexts to CC

By (1), (3),

$$\Delta(\mathit{fvar}_1) = s \tag{5}$$

By the rules for translation,

$$[\![\textsf{sign}(t_1, t_2)]\!]_\Delta = (\mathit{fvar}_1, \cdot) \tag{6}$$

The remaining cases are straightforward. □

Lemma 28 (CC Typing Weakening)**.** *If $\Gamma_1, \Gamma_2 \vdash^{CC} t : s$, and $\mathit{wf}(\Gamma_1, \Gamma', \Gamma_2)$, then $\Gamma_1, \Gamma', \Gamma_2 \vdash^{CC} t : s$.*

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

Lemma 29 (CC well-formed term gives well-formed environment)**.** *If $\Gamma \vdash^{CC} t : s$ then $\mathit{wf}(\Gamma)$.*

Proof. By induction on the typing derivation. □

Lemma 30 (CC well-formed term gives well-formed type)**.** *If $\Gamma \vdash^{CC} t : s$ then either $s = \Box$ or exists k such that $\Gamma \vdash^{CC} s : k$.*

Proof. By induction on the typing derivation. □

Lemma 31 (Substitution)**.** *If $\Sigma; \Gamma \vdash t_1 : k$, $\mathit{unique}(\Delta)$, $[\![t_1]\!]_\Delta = (s_1, \cdot)$ and $[\![t_2]\!]_\Delta = (s_2, \cdot)$, then $[\![\{t_2/x\}t_1]\!]_\Delta = (\{s_2/x\}s_1, \cdot)$.*

Proof. By induction on the structure of t_1.

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

By assumption,

$$[\![t_2]\!]_\Delta = (s_2, \cdot) \tag{1}$$

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

$$[\![\textsf{sign}(t, p)]\!]_\Delta = (\mathit{fvar}, \cdot) \tag{3}$$

By the definition of translation, (3),

$$[\![p]\!]_\Delta = (s, \cdot) \tag{4}$$

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

By inversion on (2),

$$\Sigma; \cdot \vdash p : \textsf{Prop} \tag{6}$$

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

$$\{t_2/x\}p = p \tag{8}$$

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

$$[\![\{t_2/x\}(\textsf{sign}(t, p))]\!]_\Delta = (\mathit{fvar}, \cdot) = (\{s_2/x\}\mathit{fvar}, \cdot) \tag{9}$$

The remaining cases are straightforward. □

Lemma 32 (Correctness of Translation). *1. If $\Sigma \vdash \diamond$, $[\![\Sigma]\!] = \Sigma_1$, then wf$(\Sigma_1)$ and unique(Σ_1).*

2. If $\Sigma \vdash \Gamma$, $[\![\Sigma]\!] = \Sigma_1$, $[\![\Gamma]\!]_{\Sigma_1} = (\Gamma_1, \Delta)$, then wf$(\Sigma_1, \Delta, \Gamma_1)$ and unique(Σ_1, Δ).

3. If $\mathcal{E} :: \Sigma; \Gamma \vdash t : s$, $[\![\Sigma]\!] = \Sigma_1$, $[\![\Gamma]\!]_{\Sigma_1} = (\Gamma_1, \Delta_1)$, wf$(\Sigma_1, \Delta_1, \Delta_2, \Gamma_1)$, unique$(\Sigma_1, \Delta_1, \Delta_2)$, $[\![t]\!]_{\Sigma_1, \Delta_1, \Delta_2} = (t_1, \Delta_3)$, then $\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1 \vdash^{CC} t_1 : s_1$, and $[\![s]\!]_{(\Sigma_1, \Delta_2, \Delta_3)} = (s_1, \cdot)$ and unique$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)$.

4. If $\mathcal{E} :: \Sigma; \Gamma \vdash t$, $[\![\Sigma]\!] = \Sigma_1$, $[\![\Gamma]\!]_{\Sigma_1} = (\Gamma_1, \Delta_1)$, wf$(\Sigma_1, \Delta_1, \Delta_2, \Gamma_1)$, unique$(\Sigma_1, \Delta_1, \Delta_2)$, $[\![t]\!]_{\Sigma_1, \Delta_1} = (t_1, \Delta_3)$, then $\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1 \vdash^{CC} t_1 : */\Box$ (read as t_3 is classified by $*$ or $\Box$), and unique$(\Sigma_1, \Delta_1, \Delta_2, \Delta_3)$.

Proof. In the proof of 1 and 2, we use 3 only when Σ or Γ is smaller.

1. By induction on the structure of (Σ).

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

By assumption,

$$\Sigma', a : t \vdash \diamond \tag{1}$$

$$[\![\Sigma', a : t]\!] = (\Sigma_1, \Delta, a : s) \tag{2}$$

$$\text{where } [\![\Sigma']\!] = \Sigma_1 \tag{3}$$

$$\text{and}[\![t]\!]_{\Sigma_1} = (s, \Delta) \tag{4}$$

By inversion of (1),

$$\Sigma' \vdash \diamond \tag{5}$$

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

By I.H. on Σ',

$$\textit{wf}(\Sigma_1) \text{ and } \textit{unique}(\Sigma_1) \tag{7}$$

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

$$\Sigma_1, \Delta \vdash^{CC} s : \Box \text{ and } \textit{unique}(\Sigma_1, \Delta) \tag{8}$$

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

$$\textit{wf}(\Sigma_1, \Delta, a : s), \text{ and } \textit{unique}(\Sigma_1, \Delta, a : s) \tag{9}$$

2. By induction on the structure of Γ.

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

By assumption,

$$\Sigma \vdash \Gamma', x : t \tag{1}$$

$$[\![\Sigma]\!] = \Sigma_1 \tag{2}$$

$$[\![\Gamma', x : t]\!]_{\Sigma_1} = ((\Gamma_1, x : s), (\Delta_1, \Delta_2, \Delta_3)) \tag{3}$$

$$\text{where } [\![\Gamma']\!]_{\Sigma_1} = (\Gamma_1, \Delta_1) \tag{4}$$

$$\text{and } \textit{wf}(\Sigma_1, \Delta_1, \Delta_2), \textit{unique}(\Sigma_1, \Delta_1, \Delta_2) \tag{5}$$

$$\text{and } [\![t]\!]_{\Sigma_1, \Delta_1, \Delta_2} = (s, \Delta_3) \tag{6}$$

By inversion of (1),

$$\Sigma \vdash \Gamma' \tag{7}$$

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

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

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

$$\text{and } \textit{unique}(\Sigma_1, \Delta_1, \Delta_2, \Delta_3) \tag{10}$$

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

$$\textit{wf}(\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1, x : s) \tag{11}$$

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

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

By assumption,

$$\mathcal{E} = \frac{\mathcal{E}' :: \Sigma \vdash \Gamma}{\Sigma; \Gamma \vdash \textsf{Prop} : \textsf{Kind}^{\mathbf{P}}} \tag{1}$$

$$[\![\Sigma]\!] = \Sigma_1 \tag{2}$$

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

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

By *ax* rule,

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

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

$$\textit{wf}(\Sigma_1, \Delta_1, \Gamma_1) \text{ and } \textit{unique}(\Sigma_1, \Delta_1) \tag{6}$$

By Lemma weakening,

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

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

By assumption,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1 :: \Sigma; \Gamma \vdash t_1 : (\textsf{Kind}^{\mathbf{P}}, \textsf{Type}, \textsf{Prop}) \\ \mathcal{E}_2 :: \Sigma; \Gamma, x : t_1 \vdash t_2 : k_2 \qquad k_2 \in \{\textsf{Kind}^{\mathbf{P}}, \textsf{Prop}\}\end{array}}{\Sigma; \Gamma \vdash (x{:}t_1) \to\ t_2 : k_2} \tag{1}$$

$$[\![\Sigma]\!] = \Sigma_1 \tag{2}$$

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

$$\textit{wf}(\Sigma_1, \Delta_1, \Delta_2) \text{ and } \textit{unique}(\Sigma_1, \Delta_1, \Delta_2) \tag{4}$$

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

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

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

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

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

$$\text{and } \mathit{unique}(\Sigma_1, \Delta_1, \Delta_2, \Delta_3) \tag{9}$$

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

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

By Lemma 29, (8),

$$\mathit{wf}(\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1) \tag{11}$$

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

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

$$\text{and } \mathit{unique}(\Sigma_1, \Delta_1, \Delta_3, \Delta_4) \tag{13}$$

By Π, (8), (12),

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

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

By assumption,

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

$$[\![\Sigma]\!] = \Sigma_1 \tag{2}$$

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

$$\mathit{wf}(\Sigma_1, \Delta_1, \Delta_2), \mathit{unique}(\Sigma_1, \Delta_1, \Delta_2) \tag{4}$$

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

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

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

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

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

$$\text{and } \mathit{unique}(\Sigma_1, \Delta_1, \Delta_2, \Delta_3) \tag{9}$$

By (7), (8), Lemma 28 weakening,

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

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

$$\mathit{wf}(\Sigma_1, \Delta_1, \Gamma_1) \tag{11}$$

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

$$\mathit{wf}(\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1) \tag{12}$$

By Lemma 28 weakening, (12), (10),

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

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

By assumption,

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

$$[\![\Sigma]\!] = \Sigma_1 \tag{2}$$

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

$$\mathit{wf}(\Sigma_1, \Delta_1, \Delta_2), \mathit{unique}(\Sigma_1, \Delta_1, \Delta_2) \tag{4}$$

$$[\![\textsf{sign}(t_1, t_2)]\!]_{\Sigma_1, \Delta_1, \Delta_2} = (\mathit{fvar}_1, (\Delta_3, \mathit{fvar}_1 : s_2)) \tag{5}$$

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

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

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

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

$$\text{and } \mathit{unique}(\Sigma_1, \Delta_1, \Delta_2, \Delta_3) \tag{9}$$

By *var* rule, (8),

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

By (9), (7),

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

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

$$\mathit{wf}(\Sigma_1, \Delta_1, \Gamma_1) \tag{12}$$

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

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

By Lemma 28 weakening, (13), (10),

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

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

By assumption,

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

$$[\![\Sigma]\!] = \Sigma_1 \tag{2}$$

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

$$\textit{wf}(\Sigma_1, \Delta_1, \Delta_2), \textit{unique}(\Sigma_1, \Delta_1, \Delta_2) \tag{4}$$

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

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

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

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

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

$$\text{and } (k, \cdot) = [\![(x{:}u_2) \to\ u]\!]_{\Sigma_1,\Delta_1,\Delta_2,\Delta_3} \tag{9}$$

$$\text{and } \textit{unique}(\Sigma_1, \Delta_1, \Delta_2, \Delta_3) \tag{10}$$

By (8), and Lemma 29,

$$\textit{wf}(\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Gamma_1) \tag{11}$$

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

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

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

$$\text{and } \textit{unique}\Sigma_1, \Delta_1, \Delta_2, \Delta_3, \Delta_4 \tag{14}$$

By translation weakening Lemma 27 and (9),

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

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

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

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

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

By translation weakening Lemma 27 and (7),

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

By Lemma 31, (16), (18),

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

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

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

Special Reduction Rule:

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

Calculus of Construction extended with product dependent types is known to be strongly normalizing (Geuvers, 1995). We use **SN**(β) to denote the set of terms that are strongly normalizing under β reductions in CC; similarly, **SN**$(\beta\beta')$ is the set of terms that are strongly normalizing under the β and β' reduction rules. We demonstrate that CC augmented with β' is also strongly normalizing.

Lemma 33 (Strong normalization of $\beta\beta'$-reduction in CC)**.** *For all term $t \in$ **SN**(β), $t \in$ **SN**$(\beta\beta')$.*

Proof. We use the technique presented in Lindley's thesis (Lindley, 2005). We assign an ordering between terms as the dictionary order of a pair $(\beta(t), \delta(t))$, where $\beta(t)$ is the maximum $beta$-reduction steps of t, and $\delta(t)$ is defined as follows. $\delta(x) = 1$, $\delta(\lambda x{:}t.\, s) = \delta(s)$, $\delta(t_1\ t_2) = \delta(t_1) + 2\delta(t_2)$. We then prove that if $t \rightarrow_{\beta'} t'$ then $\beta(t') \leq \beta(t)$, by examining all possible β-reductions of t', and showing that t has an corresponding reduction that takes at least the same number of β-reduction steps as t'. Now $\delta((\lambda y{:}s.\, ((\lambda x{:}t.\, t_1)t_2))u) = \delta(t_1) + 2\delta(t_2) + 2\delta(u)$, and $\delta(\lambda x{:}t.\, t_1)((\lambda y{:}s.\, t_2)u) = \delta(t_1) + 2\delta(t_2) + 4\delta(u)$. Therefore, when $t \rightarrow_{\beta'} t'$, $(\beta(t'), \delta(t')) < (\beta(t), \delta(t))$. Thus, for all $t \in$ **SN**(β), $t \in$ **SN**$(\beta\beta')$. □

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

Lemma 34 (Simulation)**.** *If $t \rightarrow t'$, and and $[\![t]\!]_\Delta = (s, \Delta)$, $[\![t']\!]_\Delta = (s', \Delta)$, then $s \rightarrow^+_{\beta,\beta'} s'$.*

Proof. By examining all the reduction rules. □

Lemma 35 (Strong normalization)**.** *Aura$_0$ is strongly normalizing.*

Proof. By Lemma 34, and Lemma 33. A diverging path in Aura$_0$ implies a diverging path in CC. Since CC is strongly normalizing, Aura$_0$ is also strongly normalizing. □

Lemma 36. *If $s \rightarrow s'$, then $\{t/x\}s \rightarrow^* \{t/x\}s'$.*

Proof. By induction on the structure of s. □

Lemma 37. *If $t \rightarrow t'$, then $\{t/x\}s \rightarrow^* \{t'/x\}s$.*

Proof. By induction on the structure of s. □

Lemma 38 (Weak Confluence)**.** *If $t \rightarrow t_1$, $t \rightarrow t_2$, then exists t_3 such that $t_1 \rightarrow^* t_3$, and $t_2 \rightarrow^* t_3$.*

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

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

By assumption,

$$t_1 = \lambda x{:}u.\, s_1 \text{ where } s \rightarrow s_1 \tag{1}$$

$$t_2 = \lambda x{:}u.\, s_2 \text{ where } s \rightarrow s_2 \tag{2}$$

By I.H. on s,

$$\exists s_3 \text{ such that } s_1 \rightarrow^* s_3, \text{and } s_2 \rightarrow^* s_3 \tag{3}$$

By reduction rules,

$$t_3 = \lambda x{:}u.\, s_3 \text{ such that } t_1 \rightarrow^* t_3 \text{ and } t_2 \rightarrow^* t_3 \tag{4}$$

Case: $t = (\lambda x{:}u.\, s_1)s_2$, $t_1 = \{s_2/x\}s_1$, and

$t_2 = (\lambda x{:}u.\, s_1')s_2$ where $s_1 \rightarrow s_1'$.

By Lemma 37,

$$t_1 \rightarrow^* \{s_2/x\}s_1' \tag{1}$$

By reduction rules,

$$t_2 \rightarrow \{s_2/x\}s_1' \tag{2}$$

Case: $t = \textsf{bind}\ x\ =\ s_1\ \textsf{in}\ s_2$

where $s_1 = \textsf{bind}\ y\ =\ \textsf{return}@[a]u_1\ \textsf{in}\ u_2$

By assumption,

$$t_1 = \textsf{bind}\ x\ =\ \{u_1/x\}u_2\ \textsf{in}\ s_2 \tag{1}$$

$$t_2 = \textsf{bind}\ y\ =\ \textsf{return}@[a]u_1\ \textsf{in}\ \textsf{bind}\ x\ =\ u_2\ \textsf{in}\ s_2$$

$$\text{and } y \notin fv(s_2) \tag{2}$$

By reduction R-BINDT,

$$t_2 \rightarrow \textsf{bind}\ x\ =\ \{u_1/y\}u_2\ \textsf{in}\ \{u_1/y\}s_2 \tag{3}$$

By (2), (3),

$$t_2 \rightarrow \textsf{\textbf{bind}}\ x\ =\ \{u_1/y\}u_2\ \textsf{\textbf{in}}\ s_2 \tag{4}$$

Case: $t = \textsf{\textbf{bind}}\ x\ =\ s_1\ \textsf{\textbf{in}}\ s_2$

where $s_1 = \textsf{\textbf{bind}}\ y\ =\ u_1\ \textsf{\textbf{in}}\ u_2$

and $u_1 = \textsf{\textbf{bind}}\ z\ =\ w_1\ \textsf{\textbf{in}}\ w_2$

By assumption,

$t_1 = \textsf{\textbf{bind}}\ x\ =\ s'_1\ \textsf{\textbf{in}}\ s_2$

where $s'_1 = \textsf{\textbf{bind}}\ z\ =\ w_1\ \textsf{\textbf{in}}\ \textsf{\textbf{bind}}\ y\ =\ w_2\ \textsf{\textbf{in}}\ u_2$ (1)

$t_2 = \textsf{\textbf{bind}}\ y\ =\ u_1\ \textsf{\textbf{in}}\ \textsf{\textbf{bind}}\ x\ =\ u_2\ \textsf{\textbf{in}}\ s_2$

By applying R-BINDC rule many times,

$$t_1 \rightarrow^* t_3 \tag{2}$$

$$t_2 \rightarrow^* t_3 \tag{3}$$

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

The remaining cases are straightforward. □

Appendix B

Formal Aura language definitions

This appendix provides definitions elided from Chapter 3.

Values and applied values

Values and applied values are defined as shown in Figure B.1. In addition to standard call-by-name values some, proof-, type-, and higher-level constructs—for instance **prin** and **Type**—are considered values. Such objects can never reduce, so it reasonable to consider them values. Usefully, this allows applications (see rule WF-TM-APP) to treat uniformly different sorts of objects.

Aura's signatures

Syntax

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

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

$$\boxed{\textit{val}\ \mathrm{t}}$$

$$\frac{}{\textit{val}\ \textsf{\textbf{Type}}}\ \textsc{v-Type} \qquad \frac{}{\textit{val}\ \textsf{\textbf{Prop}}}\ \textsc{v-Prop} \qquad \frac{}{\textit{val}\ \textsf{\textbf{Kind}}}\ \textsc{v-Kind} \qquad \frac{}{\textit{val}\ x}\ \textsc{v-Var}$$

$$\frac{}{\textit{val}\ \lambda x\!:\!t_1.\ t_2}\ \textsc{v-Abs} \qquad \frac{\textit{app-val}\ t}{\textit{val}\ t}\ \textsc{v-App} \qquad \frac{}{\textit{val}\ \textsf{\textbf{self}}}\ \textsc{v-Self} \qquad \frac{}{\textit{val}\ (x\!:\!t_1) \rightarrow t_2}\ \textsc{v-Arr}$$

$$\frac{\textit{val}\ v}{\textit{val}\ \textsf{\textbf{sign}}(v, p)}\ \textsc{v-Sign} \qquad \frac{}{\textit{val}\ v\ \textsf{\textbf{says}}\ p}\ \textsc{v-Says} \qquad \frac{}{\textit{val}\ \textsf{\textbf{prin}}}\ \textsc{v-Prin} \qquad \frac{\textit{val}\ p}{\textit{val}\ \textsf{\textbf{return}}_p\ p}\ \textsc{v-PfRet}$$

$$\frac{}{\textit{val}\ \textsf{\textbf{return}}_s\ a\ p}\ \textsc{v-SaysRet} \qquad \frac{}{\textit{val}\ \textsf{\textbf{bind}}_s\ e_1\ e_2}\ \textsc{v-SaysBind}$$

$$\boxed{\textit{app-val}\ t}$$

$$\frac{}{\textit{app-val}\ \textit{ctr}}\ \textsc{av-Ctr} \qquad \frac{\textit{app-val}\ v_1 \qquad \textit{val}\ v_2}{\textit{app-val}\ v_1\ v_2}\ \textsc{av-App}$$

Figure B.1: Aura value and applied value relations

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.

Typing rules for signatures

In Figure B.2, we present the auxiliary definitions used by the main typing judgments for well-formed signatures.

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 $(x\!:\!t_1) \rightarrow t_2$, *ctrs* only appear positively in t if *ctrs* appear positively in t_2 but do not appear in t_1.

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

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

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

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

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

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

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

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

$$\frac{\textit{get\_tctr\_defns}(\textit{defns}) = \textit{defns}'}{\textit{get\_tctr\_defns}(\textit{defns}\ \textbf{with data}\ c : t\ \{\textit{cdecls}\}) \qquad = \textit{defns}'\ \textbf{with data}\ c : t\ \{\cdot\}}$$

Figure B.2: Auxiliary Definitions

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	$P; S_1; S_2 \vdash \textit{defns}$
Well-formed type constructors	$S \vdash \textit{defns} : t$
Well-formed data constructors	$P; S_1; S_2; \textit{ctr}; t \vdash \textit{cdecls}$

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

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

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

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

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

$$P; S_1; S_2 \vdash \cdot \quad \textsc{Wf-bundle-ctr-nil}$$

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

Figure B.3: Aura signature typing rules

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 $P; S_1; S_2; \mathit{ctr}; t \vdash \mathit{cdecls}$ 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 $P; S_1; S_2 \vdash \mathit{defns}$ checks that a definition is well-formed. S_1, S_2, and P have the same meaning as above. Judgment $S \vdash \mathit{defns} : t$ 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 B.3. 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

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

$$\frac{S; S; \cdot \vdash t : \textsf{Kind}}{S \vdash \cdot : t}\ \textsc{Wf-tctr-nil}$$

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

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

$$\cdot \vdash \diamond\ \textsc{Wf-sig-nil}$$

$$\frac{\begin{array}{c} S \vdash \diamond \\ S; \cdot \vdash t : \textsf{Kind} \qquad t = (x_1 : t_1) \to \cdots (x_n : t_n) \to \textsf{Prop} \qquad \mathit{ctr} \notin \mathit{dom}(S) \end{array}}{S, \textsf{assert}\ \mathit{ctr} : t \vdash \diamond}\ \textsc{Wf-assert}$$

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

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

Aura signature typing rules (cont.)

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

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

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

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

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

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

Figure B.4: Aura environment typing rules

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.

Summary of Typing Rules

Environment typing rules The typing rules for environments are in Figure B.4. 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 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.

Term typing rules We summarize the term typing rules in Figures B.5 and B.6. As we mentioned earlier, the typing judgment for terms needs to take two signature arguments. The typing rules for terms presented in Section 3.3 is a simplified version and only takes one signature argument. However, most of simplified rules can be made precise just by adding a second signature to the typing judgment. 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

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

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

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

$$\frac{\begin{array}{c} S_1; S_2; E \vdash t : k \qquad S_1; S_2; E, x\!:\!t \vdash u : k_1 \\ S_1; S_2; E \vdash (x\!:\!t) \to k_1 : k_2 \qquad k \in \{\textsf{\textbf{Type}}, \textsf{\textbf{Prop}}, \textsf{\textbf{Kind}}\} \qquad k_2 \in \{\textsf{\textbf{Type}}, \textsf{\textbf{Prop}}\} \end{array}}{S_1; S_2; E \vdash \lambda x\!:\!t.\ u : (x\!:\!t) \to k_1}\ \textsc{Wf-tm-abs}$$

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

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

Figure B.5: Aura typing rules, extended, functional programing

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 B.7. 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

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

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

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

$$\frac{\begin{array}{c} S_1; S_2; E \vdash e_1 : a\ \textsf{\textbf{says}}\ P \\ S_1; S_2; E \vdash e_2 : (x{:}P) \rightarrow a\ \textsf{\textbf{says}}\ Q \qquad x \notin \mathit{fv}(Q) \end{array}}{S_1; S_2; E \vdash \textsf{\textbf{bind}}_s\ e_1\ e_2 : a\ \textsf{\textbf{says}}\ Q}\ \textsc{Wf-tm-says-bind}$$

$$\frac{S_1; S_2; \cdot \vdash a : \textsf{\textbf{prin}} \qquad S_1; S_2; \cdot \vdash P : \textsf{\textbf{Prop}}}{S_1; S_2; E \vdash \textsf{\textbf{sign}}(a, P) : a\ \textsf{\textbf{says}}\ P}\ \textsc{Wf-tm-sign}$$

$$\frac{S_1; S_2; E \vdash P : \textsf{\textbf{Prop}}}{S_1; S_2; E \vdash \textsf{\textbf{say}}\ P : \textsf{\textbf{pf}}\ \textsf{\textbf{self}}\ \textsf{\textbf{says}}\ P}\ \textsc{Wf-tm-say} \qquad\qquad \frac{S_1; S_2; E \vdash P : \textsf{\textbf{Prop}}}{S_1; S_2; E \vdash \textsf{\textbf{pf}}\ P : \textsf{\textbf{Type}}}\ \textsc{Wf-tm-Pf}$$

$$\frac{S_1; S_2; E \vdash p : P \qquad S_1; S_2; E \vdash P : \textsf{\textbf{Prop}}}{S_1; S_2; E \vdash \textsf{\textbf{return}}_p\ p : \textsf{\textbf{pf}}\ P}\ \textsc{Wf-tm-pf-ret}$$

$$\frac{S_1; S_2; E \vdash e_1 : \textsf{\textbf{pf}}\ P \qquad S_1; S_2; E \vdash e_2 : (x{:}P) \rightarrow \textsf{\textbf{pf}}\ Q \qquad x \notin \mathit{fv}(Q)}{S_1; S_2; E \vdash \textsf{\textbf{bind}}_p\ e_1\ e_2 : \textsf{\textbf{pf}}\ Q}\ \textsc{Wf-tm-pf-bind}$$

$$\frac{\begin{array}{c} S_1; S_2; E \vdash v_1 : k \qquad S_1; S_2; E \vdash v_2 : k \qquad \mathit{atomic}\ S_2\ k \\ \mathit{val}(v_1) \qquad \mathit{val}(v_2) \qquad S_1; S_2; E, x \sim (v_1 = v_2){:}k \vdash e_1 : t \qquad S_1; S_2; E \vdash e_2 : t \end{array}}{S_1; S_2; E \vdash \textsf{\textbf{if}}\ v_1 = v_2\ \textsf{\textbf{then}}\ e_1\ \textsf{\textbf{else}}\ e_2 : t}\ \textsc{Wf-tm-if}$$

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

Figure B.6: Aura typing rules, extended, access control

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

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

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

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

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

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

Figure B.7: Aura branch set typing rules

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

```
cons → λx: nat. λxs:List nat. b
```

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

$t_c = (x{:}\textbf{Type}) \to (y{:}x) \to (z{:}\,\text{List}\ x) \to \text{List}\ x.$

$t_b = (x{:}\text{nat}) \to (xs{:}\,\text{List nat}) \to \text{nat}.$

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).

Appendix C

Mechanized Aura$_{\text{conf}}$ definitions

This appendix provides the definitions of Aura$_{\text{conf}}$'s syntax and semantics as a deep embedding in Coq. These definition are known to compile under Coq version 8.1pl3.

Aura$_{\text{conf}}$ is formalized in Coq using a locally nameless variable representation (Aydemir et al., 2008). Free variables are represented by names, and bound variables by de Bruijn indices (de Bruijn, 1972). Some typing rules, like abstraction, need to locally treat a bound variable as free. The variable's new name may be selected from any co-finite subset of names. This representation provides for convenient induction principals.

One nonstandard aspect of these definitions is the handling of built-in operators, such as **bind** e1 e2. Instead of adding special productions to the syntax, we treated **bind** e1 e2 as two applications written in Coq as tm_app (tm_app TM_PF_BIND e1) e2. Though initially appealing this decision was problematic. When inverting the type derivation of a **bind** expression, one must rule out the case that tm_app TM_PF_BIND e1 is applied to e2 via the application rule. Such uninteresting inversions result in substantial overhead in the proofs. We introduced an alternative *meta-application* operator, tm_mapp, to avoid this problem with constructs added later in proof development.

This development is heavily based on the Coq scripts used to define Aura$_0$ and core Aura. Limin Jia, in particular, did a tremendous job refining these language definitions and developing their metatheory.

C.1 Syntax

```
(*** An encoding of the core language in Pure−Type−Systems−style, using
 *   multiple typing judgments to differentiate syntactic classes.
 *
 *   This approach is inspired by the PTS work and by the Henk intermediate
 *   language proposed by Simon Peyton Jones and Erik Meijer.
 *)


(** Terms −− these are the common datatype on which types, expressions,
 * kinds, etc. are built.
 *)

Definition bv  := nat.
Definition con := nat.
Definition ctr :=  prod nat  nat.
(* the second nat indicate the number of params*)
(* e.g. cons = (n, 1), because List: Type → Type*)

Inductive principal_const: Set :=
  pc_intro : string → principal_const.

Inductive tm : Set :=
  | tm_fv    : atom →tm              (* free variables *)
  | tm_bv    : bv → tm               (* bound variables *)
  | tm_con   : con → tm              (* built−in constants like Type, Prop, says, sign, or IO *)
  | tm_ctr   : ctr → tm              (* user defined constructors, like Bool and List *)
  | tm_app   : tm → tm → tm          (* applications like (e1 e2) or e1[t]*)
  | tm_mapp : tm →tm → tm            (* meta application not typable using arr rule (experimental) *)
  | tm_prin  : principal_const → tm        (* principals *)
  | tm_abs   : tm → tm → tm → tm          (* abstractions \ (x:t).e   or  /\ (a::K).e *)
  | tm_arr   : tm → tm → tm → tm          (* dependent products (x:t) → e with a latent effect
      annotation *)
  | tm_mat   : tm → tm → tm → tm          (* single−level pattern matching *)
  | tm_brn   : ctr → tm → tm → tm         (* branch in the case construct *)
  | tm_cast  : tm → tm → tm → tm          (* cast(e,t, reason) : t *)
  | tm_enc   : tm → tm → nat → tm         (* E(p, e, n) *)
  | tm_letat   : tm → tm → tm → tm        (* let x at w = e1 in e2 *)
  | tm_forbnd : tm → tm → tm → tm         (* bind x at w = e1 in e2 *).

Inductive world : Set :=
  | w_prin : tm → world
  | w_bot : world
  | w_top : world.
```

C.2 Environments

```
Inductive bnd : Set :=
| bnd_var : tm → world → bnd
| bnd_eq  : tm → tm → tm → world → bnd
.

Notation ”a ˜: t @ U” := (a ˜ bnd_var t U)
  (at level 31, left associativity ) : env_scope.

Definition binds_var x t V E :=
  binds x (bnd_var t V) E.

Definition binds_eq x e1 e2 t V E :=
  binds x (bnd_eq e1 e2 t V) E.

Definition env := env bnd.

Inductive std_bound_var (E: env) (x: atom): Prop :=
  sbv: forall t V, binds_var x t V E → std_bound_var E x.
```

C.3 Constants and Worlds

```
(* Case list terminator *)
Definition TM_NONE := tm_con 00.

(* Sort constants *)
Definition S_KIND := tm_con 1. (* Box i.e.  Type : S_KIND and Prop : S_KIND *)

(* Kind constants *)
Definition K_TYPE := tm_con 10.
Definition K_PROP := tm_con 11.

(* Prop constants *)
Definition P_SAYS := tm_con 20.
Definition P_ISA  :=  tm_con 21.

(* Type constants *)
Definition T_PRIN := tm_con 30.
Definition T_PF   :=  tm_con 31.
Definition T_FOR := tm_con 32.
Definition T_BITS := tm_con 33.

(* Expr constants *)
Definition TM_PF_RETURN := tm_con 40.
Definition TM_PF_BIND   :=  tm_con 41.
Definition TM_SAY       :=  tm_con 42.
Definition TM_PIF       :=  tm_con 43.
Definition TM_FOR_RET   :=  tm_con 45.
Definition TM_FOR_BIND := tm_con 46.
Definition TM_FOR_RUN  :=  tm_con 47.
Definition TM_AS_BITS   :=  tm_con 48.

(* Proof constants *)
Definition PF_SAYS_RETURN := tm_con 50.
Definition PF_SAYS_BIND := tm_con 51.
Definition PF_SIGN      :=  tm_con 52.

(* Meta-terms *)
Definition STM_FAIL     :=  tm_con 60.
Definition STM_CONV_CAST := tm_con 61.
Definition STM_TRUE_CAST := tm_con 62.
Definition STM_JUST_CAST := tm_con 63.
Definition STM_WORLD_BOT := tm_con 64.
Definition STM_WORLD_PRIN := tm_con 65.
Definition STM_WORLD_TOP := tm_con 66.
```

```
(***********************************************
 * Simple Worlds
 *)

(* Simple world represent ”physical” keys *)
Inductive simple_world: world → Prop :=
| sw_bot: simple_world w_bot
| sw_const: forall p, simple_world (w_prin (tm_prin p)).

(***********************************************
 * More worlds
 *)

Definition world_to_tm (W: world): tm :=
  match W with
  | w_bot ⇒ STM_WORLD_BOT
  | w_prin p ⇒ tm_mapp STM_WORLD_PRIN p
  | w_top ⇒ STM_WORLD_TOP
  end.

Inductive world_lteq: world → world → Prop :=
   wl_refl  : forall (W: world), world_lteq W W
| wl_bot   : forall (W: world), world_lteq w_bot W
| wl_top   : forall (W: world), world_lteq W w_top.
```

C.4 Values

```
(*********************************************
 * Value forms
 *)

(* Note:

   This definition must change whenever new constants
   are added to the language.
    - other principal constants
    - built-in values like 0, 1, 2, "foo", etc.
*)

Inductive value : tm → Prop :=
| value_TYPE: value K_TYPE
| value_PROP: value K_PROP
| value_KIND: value S_KIND
| value_var : forall x, value (tm_fv x)
| value_abs : forall t e U,
              value (tm_abs t e U)
| value_app : forall t,
              applied_value t → value t
| value_prin : forall p, value (tm_prin p)

| value_arr : forall t1 t2 U, value (tm_arr t1 t2 U)

| value_sign: forall v p, value v → value (tm_app (tm_app PF_SIGN v)p)

| value_P_SAYS: forall v p, value (tm_app (tm_app P_SAYS v) p)
| value_T_PRIN: value T_PRIN
| value_T_BITS: value T_BITS
| value_T_FOR: forall a t, value (tm_mapp (tm_mapp T_FOR a) t)
| value_T_ISA: forall e t, value (tm_mapp (tm_mapp P_ISA e) t)
| value_T_PF: forall P, value (tm_app T_PF P)

(* We don't care whether p contains redexes, since it is a proof *)
| value_TM_PF_RETURN : forall p,
                value p → value (tm_app TM_PF_RETURN p)
| value_PF_SAYS_RETURN: forall a p,
               value (tm_app (tm_app PF_SAYS_RETURN a) p)
| value_PF_SAYS_BIND: forall e1 e2, value (tm_app (tm_app PF_SAYS_BIND e1) e2)
| value_enc: forall e1 e2 n, value (tm_enc e1 e2 n)
| value_cast_true: forall v t,
                     value v →
                     value (tm_cast v t STM_TRUE_CAST)
| value_cast_just: forall v1 t v2,
                     value v1 →
                     value v2 →
                     value (tm_cast v1 t (tm_mapp STM_JUST_CAST v2))
```

```
(* applied_values may safely use the tm_app construct without
   introducing any evaluation *)
with applied_value : tm → Prop :=
| applied_value_ctr : forall c, applied_value (tm_ctr c)
| applied_value_app : forall t1 t2,
                          applied_value t1
                       → value t2
                       → applied_value (tm_app t1 t2)
.
```

C.5 Type Signatures

```
(∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗
 ∗ Signatures
 ∗)

(∗∗ Datatype definitions :

    A signature [sig] is a list of bundles of mutually recursive Type (or Prop) declarations.

    A bundle [bundle] is either a list of datatype definitions [data_decl] or an assertion

    Each data declaration contains:
       − a Type (or Prop) constructor [ tctr ] identifier .
       − the kind of the Type/Prop being declared
       − a list of constructor [ ctr ] declarations [ ctr_decls ]

    CONVENTION: We use the notation [tctr] to refer to a Type or Prop constructor identifier .
                We use the notation [ctr] to refer to an exp or proof constructor identifier .
                Both [ tctr ] and [ctr] values have Coq type [ctr] (both ranged over by [c])
 ∗)

(∗ Definition of datatype declarations in the context ∗)
Definition ctr_decls  := list (ctr ∗ tm).
Definition data_decl  := (ctr ∗ tm ∗ (option ctr_decls))%type.
Definition defns      := list data_decl.
Inductive bundle : Set :=
| bundle_defns  : defns → bundle
| bundle_assert : ctr → tm → bundle
.
Definition sig        := list (bundle).


(∗ Utilities for working with signatures ∗)

Definition dom_ctr_decls(L:ctr_decls) :=
List.map (fun (x:ctr∗tm) ⇒ let (n,_) := x in n) L.

Fixpoint ctr_decls_lookup (c:ctr) (cdl:ctr_decls) {struct cdl}: option tm :=
  match cdl with
    | nil          ⇒ None
    | (c’, t) :: cdl ⇒
        if (eq_ctr_dec c c’) then Some t
                          else ctr_decls_lookup c cdl
  end.

Fixpoint bundle_defns_ctr_lookup (c:ctr) (defns: defns) {struct defns}:
    option tm :=
  match defns with
    | nil              ⇒ None
    | (c', t, None)::ds ⇒
       if (eq_ctr_dec c c')  then Some t
         else bundle_defns_ctr_lookup c ds
    | (c', t, Some cdl)::ds ⇒
        if (eq_ctr_dec c c')  then Some t
                      else match ctr_decls_lookup c cdl with
                             | None ⇒bundle_defns_ctr_lookup c ds
                             | Some t ⇒Some t
                            end
  end.

(* Gets the kind/type associated with constructor c in signature S
   Returns:
             [None] if c isn't in S
             [Some t] if c's kind/type is t in S
*)
Fixpoint sig_lookup_ctr (c:ctr) (S:sig) {struct S}: option tm :=
  match S with
    | nil    ⇒ None
    | b::bs ⇒ match b with
                 | bundle_defns defns ⇒
                   match bundle_defns_ctr_lookup c defns with
                     | Some t ⇒Some t
                     | None ⇒sig_lookup_ctr c bs
                   end
                 | bundle_assert _ _ ⇒ sig_lookup_ctr c bs
              end
  end.

(* Gets the kind/type associated with an assertion c in signature S
   returns: [None] if c isn't in S
             [Some t] if c's kind/type is t in S
*)
Fixpoint sig_lookup_assn (c:ctr) (S:sig) {struct S}: option tm :=
  match S with
    | nil    ⇒ None
    | b::bs ⇒ match b with
                 | bundle_defns defns ⇒sig_lookup_assn c bs
                 | bundle_assert c' t ⇒
                   if eq_ctr_dec c c' then Some t else sig_lookup_assn c bs
              end
  end.


Definition ctr_decls_has (c:ctr) (t:tm) (cdl:ctr_decls) :=
ctr_decls_lookup c cdl = Some t.

Definition defns_has_ctr (c:ctr) (tp: tm) (d:defns) :=
 bundle_defns_ctr_lookup c d = Some tp.
```

```
Definition sig_has_ctr (c:ctr) (t:tm) (S:sig) : Prop :=
  sig_lookup_ctr c S = Some t.

Definition sig_has_assn (c:ctr) (t:tm) (S:sig) : Prop :=
  sig_lookup_assn c S = Some t.

Fixpoint get_defns_ctrs_of (c:ctr) (d:defns) {struct d} : option (list ctr) :=
  match d with
    | nil ⇒ None
    | (c', t, cdl) :: ds ⇒
      if eq_ctr_dec c c'
        then match cdl with
               | None ⇒None
               | Some cdl' ⇒Some (dom_ctr_decls cdl')
             end
        else get_defns_ctrs_of c ds
  end.

(* Gets the list of expression constructors for type c from the signature S.
   Returns: [None] if c is temporarily undefined in S
            [Some None] if c is not defined in
            [Some l], where l is the list of term constructors, otherwise
*)
Fixpoint get_ctrs_of (c:ctr) (S:sig) {struct S}: option (list ctr) :=
  match S with
    | nil    ⇒ None
    | b::bs ⇒ match b with
                 | bundle_defns d ⇒
                   match get_defns_ctrs_of c d with
                     | Some l ⇒Some l
                     | None ⇒get_ctrs_of c bs
                   end
                 | bundle_assert _ _ ⇒ get_ctrs_of c bs
               end
  end.

Definition get_tctr_defns_of (d:defns) : defns :=
  let strip x :=
    match x with (n,t,_) ⇒ (n,t,None) end
  in
  List.map strip d.

Fixpoint dom_defns (d:defns) {struct d}: list ctr :=
  match d with
  | nil ⇒ nil
  | (c,_,None)::ds ⇒ c ::(dom_defns ds)
  | (c,_,Some cdl)::ds ⇒ c ::(dom_ctr_decls cdl)++(dom_defns ds)
  end.

Fixpoint dom_bundle (b:bundle) : list ctr :=
match b with
  | bundle_defns d ⇒ dom_defns d
  | bundle_assert c t ⇒ c :: nil
end.

Definition dom_sig (S: sig) : list ctr :=
  List.flat_map (dom_bundle) S.

Inductive is_tm_arr : tm → Prop :=
| is_tm_arr_arr : forall t1 t2 U, is_tm_arr (tm_arr t1 t2 U)
.

(* [constructs tm s] is intended to hold when [tm] has form
     [t1 → ... → tn → s] and s is not an arrow *)
Inductive constructs : tm → tm → Prop :=
| constructs_base : forall s, ~( is_tm_arr s) → constructs s s
| constructs_arr : forall s t1 t2,
    constructs t2 s → constructs (tm_arr t1 t2 STM_WORLD_BOT) s.

Inductive fully_applied_args : tm → ctr → list tm → Prop :=
| fully_applied__args_ctr : forall c, fully_applied_args (tm_ctr c) c nil
| fully_applied_args_app : forall t tn c args,
    fully_applied_args t c args →
    fully_applied_args (tm_app t tn) c (tn :: args)
.

Inductive fully_applied_typ : tm → ctr → tm → Prop :=
| fully_applied_TYPE : forall c, fully_applied_typ (tm_ctr c) c K_TYPE
| fully_applied_PROP : forall c, fully_applied_typ (tm_ctr c) c K_PROP
| fully_applied_typ_app : forall k1 k2 t tn c,
    fully_applied_typ t c k2 →
    fully_applied_typ (tm_app t tn) c (tm_arr k1 k2 STM_WORLD_BOT)
.

(*
 * All_tm_bv holds when the list of terms consists entirely of bound variables.
 *)

(* The ordering of bound variables are relevant*)
Inductive all_tm_bv : nat → ( list tm) → Prop :=
| all_tm_bv_nil : forall n, all_tm_bv n nil
| all_tm_bv_cons : forall l n, all_tm_bv (n+1) l → all_tm_bv n ((tm_bv n) :: l).

(* has_level t n if t = t1 → ... → tn → s*)
Inductive has_level: tm → nat → Prop:=
| has_level_base: forall u, ~is_tm_arr u → has_level u 0
| has_level_arr : forall u n t U,
    has_level u n → has_level (tm_arr t u U) (1+n).

(* ctrs_of t = l where l is a list of (posibly non−unique)
  nats where each n in l occurence in t as (tm_ctr n) *)
Fixpoint ctrs_of (t:tm) : list ctr :=
  match t with
  | tm_fv _           ⇒ nil
  | tm_bv _           ⇒ nil
  | tm_con _          ⇒ nil
  | tm_ctr c          ⇒ c :: nil
  | tm_app t1 t2      ⇒ (ctrs_of t1) ++ (ctrs_of t2)
  | tm_mapp t1 t2     ⇒ (ctrs_of t1) ++ (ctrs_of t2)
  | tm_prin _         ⇒ nil
  | tm_abs t1 t2 U    ⇒ (ctrs_of t1) ++ (ctrs_of t2) ++ (ctrs_of U)
  | tm_arr t1 t2 U    ⇒ (ctrs_of t1) ++ (ctrs_of t2) ++ (ctrs_of U)
  | tm_mat t1 t2 t3   ⇒ (ctrs_of t1) ++ (ctrs_of t2) ++ (ctrs_of t3)
  | tm_brn c t2 t3    ⇒ (ctrs_of t2) ++ (ctrs_of t3)
  | tm_cast t1 t2 t3    ⇒ (ctrs_of t1) ++ (ctrs_of t2) ++ (ctrs_of t3)
  | tm_enc _ _ _        ⇒ nil (* encryptions are opaque *)
  | tm_letat t1 t2 t3   ⇒ (ctrs_of t1) ++ (ctrs_of t2) ++ (ctrs_of t3)
  | tm_forbnd t1 t2 t3  ⇒ (ctrs_of t1) ++ (ctrs_of t2) ++ (ctrs_of t3)
  end.

Definition disjoint (L1 L2: list ctr) := forall n, (In n L1) → ˜(In n L2).

(* [positive (n_1, ... n_m) t] is intended to hold
  when construtor application of [n_i] appear at most
   positively in [t]. As definied this judgment is very
  conservative *)
Inductive positive (ctrs: list ctr): tm → Prop :=
|  positive_arr : forall t1 t2 U,
       disjoint ctrs (ctrs_of t1)
  → positive ctrs t2
  → positive ctrs (tm_arr t1 t2 U)
| positive_app: forall t1 t2,
       positive ctrs t1
  →  disjoint ctrs (ctrs_of t2)
  → positive ctrs (tm_app t1 t2)
|  postive_ctr : forall n,
       positive ctrs (tm_ctr n).

Inductive branches_cover_aux : (list ctr) → tm → Prop :=
| bca_nil  : branches_cover_aux nil TM_NONE
| bca_cons : forall c l t branches,
             branches_cover_aux l branches →
             branches_cover_aux (c::l) (tm_brn c t branches).

(* [branches_cover S branches ctr] is intended to hold when [branches] contains exactly
  one [tm_brn] per term constructor which creates a [ctr]. *)
Definition branches_cover (S:sig) (branches:tm) (ctr: ctr): Prop :=
  exists l, get_ctrs_of ctr S = Some l /\ branches_cover_aux l branches.
```

C.6 Conversion Relation

```
(∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗
 ∗ Conversion relation
 ∗)

(∗ Note: E may contain conversion assumptions of
 ∗ the form:     e1 = e2 :  t
 ∗)
(∗ The world here is a dynamic world, which is used to
 ∗ index definitional equalities (x: t1 = t2 @ W) in the
 ∗ environment. Perhaps this is not currently useful.
 ∗ It would be useful if conversion included full beta−reduction ∗)
Inductive converts: env → world → tm → tm → Prop :=
(∗ equivalence axioms ∗)
| conv_refl   : forall E w t, lc t → converts E w t t

| conv_sym   : forall E w t1 t2,
                 converts E w t1 t2
               → converts E w t2 t1

| conv_trans : forall t3 E w t1 t2,
                 converts E w t1 t2
               → converts E w t2 t3
               → converts E w t1 t3

(∗ definitional equality ∗)
| conv_axiom : forall E w t1 t2 k V x,
                 binds_eq x t1 t2 k V E
               → world_lteq V w
               → converts E w t1 t2

(∗ congruence axioms ∗)
| conv_app   : forall E w t1 t2 u1 u2,
                 converts E w t1 u1
               → converts E w t2 u2
               → converts E w (tm_app t1 t2) (tm_app u1 u2)

| conv_mapp : forall E w t1 t2 u1 u2,
                 converts E w t1 u1
               → converts E w t2 u2
               → converts E w (tm_mapp t1 t2) (tm_mapp u1 u2)

| conv_abs   : forall L E w t1 t2 u1 u2 U V,
                 converts E w t1 u1
               → (forall x, x\notin L → converts E w (t2^^ tm_fv x) (u2^^tm_fv x) )
               → converts E w U V
               → converts E w (tm_abs t1 t2 U) (tm_abs u1 u2 V)

| conv_arr   : forall L E w t1 t2 u1 u2 U V,
                 converts E w t1 u1
               → converts E w U V
               → ( forall x, x\notin L → converts E w (t2^^tm_fv x) (u2^^tm_fv x))
               → converts E w (tm_arr t1 t2 U) (tm_arr u1 u2 V)

| conv_letat : forall L E w e11 e12 e21 e22 U1 U2,
                 converts E w U1 U2
               → converts E w e11 e21
               → ( forall x, x \notin L →
                        converts E w (e12 ^^ tm_fv x) (e22 ^^ tm_fv x))
               → converts E w (tm_letat U1 e11 e12) (tm_letat U2 e21 e22)

| conv_forbnd : forall L E w e11 e12 e21 e22 U1 U2,
                 converts E w U1 U2
               → converts E w e11 e21
               → ( forall x, x \notin L →
                        converts E w (e12 ^^ tm_fv x) (e22 ^^ tm_fv x))
               → converts E w (tm_forbnd U1 e11 e12) (tm_forbnd U2 e21 e22)
.
```

C.7 Atomic Types

```
Inductive all_enum_decls: sig → list ctr → Prop :=
  | all_enum_decls_base: forall S, all_enum_decls S nil
  | all_enum_decls_ind1: forall S c n l, all_enum_decls S l →
     sig_has_ctr c (tm_ctr n) S →
    all_enum_decls S (c :: l).
```

```
Inductive atomic_type: sig → tm → Prop :=
 | atomic_type_prin: forall S, atomic_type S T_PRIN
 | atomic_type_ctr: forall S t l,
    get_ctrs_of t S = (Some l) →
   all_enum_decls S l →
   atomic_type S (tm_ctr t).
```

C.8 Fact Contexts

```
(*********************************************
 * Fact contexts
 *)

Inductive fact_ctx : Set :=
  | fc_empty : fact_ctx
  | fc_cons : tm → tm → fact_ctx → fact_ctx .

Inductive fctx_has: fact_ctx → tm → tm → Prop :=
  | fc_here : forall F e t, fctx_has (fc_cons e t F) e t
  | fc_there : forall F e1 t1 e2 t2,
                 fctx_has F e1 t1 → fctx_has (fc_cons e2 t2 F) e1 t1.
```

C.9 Notation for Syntax

```
(***********************************************
 * Notation for common objects
 *)

Notation ”t ’FOR’ B” := (tm_mapp (tm_mapp T_FOR t) B) (at level 68).
Notation ”t1 ’ISA’ t2” := (tm_mapp (tm_mapp P_ISA t1) t2) (at level 68).
Notation ”A ’SAYS’ P” := (tm_app (tm_app P_SAYS A) P) (at level 68).
Notation ”’PF’ P” := (tm_app T_PF P) (at level 68).
Notation ”’FORRET’ [ A , t ] e” := (tm_mapp(tm_mapp(tm_mapp TM_FOR_RET A) t) e)
                                              (at level 68).
Notation ”’FORBND’ [ A , t , e1 , e2 ]” := (tm_forbnd (t FOR A) e1 e2)
                                              (at level 68).
Notation ”’ASBITS’ e” := (tm_mapp TM_AS_BITS e) (at level 68).
Notation A := (tm_prin (pc_intro ”Alice”)).
Notation B := (tm_prin (pc_intro ”Bob”)).
Notation C := (tm_prin (pc_intro ”Charlie”)).
Notation ”’tbot’” := (world_to_tm w_bot).
Notation ”’SAY’ [ a ] p” := (tm_mapp (tm_mapp TM_SAY a) p) (at level 68).
Notation ”’FORRUN’ e” := (tm_mapp TM_FOR_RUN e) (at level 68).
Notation ”’FAIL’ p” := (tm_mapp STM_FAIL p) (at level 68).
Notation ”’WORLD’ t” := (tm_mapp STM_WORLD_BOT t) (at level 68).
Notation ”’SIGN’ [ a ] p” := (tm_app (tm_app PF_SIGN a) p) (at level 68).
```

C.10 Typing Relation

```
(*********************************************
 * Typing rules
 *)

(* Not well−formed, just good−enough approximation of typing
   to be used under ASBITS *)
Inductive ge_tm (E: env): tm → tm → Prop :=
  ge_tm_var: forall x t V, binds_var x t V E → ge_tm E (tm_fv x) t
| ge_tm_enc: forall a e n, ge_tm E (tm_enc a e n) T_BITS
| ge_tm_cast: forall e t A r,
                 value (tm_cast e (t FOR A) r) →
                 ge_tm E e T_BITS →
                 lc r →
                 lc (t FOR A) →
                 (forall x,
                    x \in (fv r) \u (fv (t FOR A)) →
                    std_bound_var E x) →
                 ge_tm E (tm_cast e (t FOR A) r) (t FOR A).

Inductive wf_env (S:sig) (F: fact_ctx ) : world → env → Prop :=

(*
  simple_world W
−−−−−−−−−−−−−−−−−−−
  S; F; W |− .
*)
| wf_env_nil          : forall W,
                          simple_world W →
                          wf_env S F W empty


(*
    S; F; W |− E
    S; F; W; E; V; U |− t : k
    x # E
    << side condition on t and k >>
    −−−−−−−−−−−−−−−−−−−−−
    S |− E & x ~: t @ V
*)
| wf_env_cons_var : forall k x t E W V,
                      wf_env S F W E          (*premise is superfluous, but
                                                avoids mutual inductions
                                                later on *)
                   → wf_world S F W E V
                   → wf_tm S F W E w_bot w_bot t k
                   → x # E
                   → (k = K_TYPE \/ k = K_PROP) \/
                     ((t = K_TYPE \/ t = K_PROP)
                         /\ V = w_bot)
                   →  wf_env S F W (E & x ~:  t  @ V)

(*  S; F;  W |− E
   S; F;  W; E; _|_  ;  U |− e1 : Prin
   S; F;  W; E; _|_;  U |− e2 : Prin
   atomic S
   val  e1,  e2
   x # E
   −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
   S; F;  w_bot |− E & x ~ e1 = e2 :   Prin
*)
| wf_env_cons_eq :  forall  x  e1 e2 t  W V U E,
                      wf_env S F W E
                   →  wf_tm S F W E w_bot U e1 t
                   →  wf_tm S F W E w_bot U e2 t
                   →  atomic_type S t
                   →  x # E
                   → value e1
                   → value e2
                   → wf_tm S F W E w_bot w_bot t K_TYPE
                   → wf_world S F W E V
                   → wf_env S F W (E & x ~ (bnd_eq e1 e2 t V))

with wf_world (S: sig)  (F:  fact_ctx )  :  world → env → world → Prop :=

(*
 *  S; F;  W |− E
 * −−−−−−−−−−−−−−−−−−−−
 *  S; F;  W; E |− bot
 *)
| wf_world_bot:  forall  W E,
    wf_env S F W E →
    wf_world S F W E w_bot

(*
 *    S; F;  W; E; bot;  bot  |− p:  prin
 *    value p
 *   −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 *    S; F;  W; E |− (p)
 *)
|   wf_world_prin :  forall  W E p,
      wf_tm S F W E w_bot w_bot p T_PRIN →
      value p →
      wf_world S F W E (w_prin p)

(*
 *  S; F;  W |− E
 * −−−−−−−−−−−−−−−−−−−−
 *  S; F;  W; E |− top
 *)
```

```
| wf_world_top:  forall  W E,
    wf_env S F W E →
    wf_world S F W E w_top

with wf_worlds (S: sig)  (F:  fact_ctx )  :  world → env → world →
        world → Prop :=
(*
 *    S; F;  W; E |− U
 *    S; F;  W; E |− V
 *    S; F;  W; E; V; U |− ok *)
|  wf_worlds_intro :  forall  W E V U,
    wf_world S F W E V →
    wf_world S F W E U →
    wf_worlds S F W E V U

with wf_tm (S:sig)  (F:  fact_ctx ) :
  world → env → world → world → tm → tm → Prop :=

(*
 *  S; F;  W; E; V; U |− ok
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 *  S; F;  W; E; V; U |− Type : Kind
 *
 *)

|  wf_tm_K_TYPE : forall W E V U,
     wf_worlds S F W E V U →
     wf_tm S F W E V U K_TYPE S_KIND

(*
 *  S;F;W;E;V;U |− ok
 * −−−−−−−−−−−−−−−−−−−−−−
 *  S;F;W;E;V;U |− Prop : Kind
 *
 *)

|  wf_tm_K_PROP : forall W E V U,
     wf_worlds S F W E V U →
     wf_tm S F W E V U K_PROP S_KIND

(*
 *  S; F;  W; E, x:u1 @w_bot | wbot ; bot;  bot|− u2: k2
 *  S; F;  W: E; wbot; wbot|− u1 : k1
 *  S; F;  W; E; V; U |− ok
 *  S; F;  W; E; V; U0 |− ok
 *  k2 = Type/Prop/Kind
 *  k1 = Type/Prop or u1 = Type/Prop
 *  S; F;  W; E |− U0
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 *  S; F;  W: E; w_bot; w_bot |− (x:k1)  −−U0→ u2 : k2
 *)
| wf_tm_arr : forall L W E u1 u2 k1 k2 V U U0,
  wf_tm S F W E w_bot w_bot u1 k1 →
  wf_worlds S F W E V U →
  wf_worlds S F W E V U0 →
  ( forall x,
      x \notin L →
      wf_tm S F W (E & x ˜: u1 @ w_bot) w_bot w_bot (u2 ˆˆ (tm_fv x)) k2) →
  k2 = K_TYPE \/ k2 = K_PROP \/ k2 = S_KIND →
  u1 = K_TYPE \/ u1 = K_PROP \/ k1 = K_TYPE \/ k1 = K_PROP →
  wf_tm S F W E V U (tm_arr u1 u2 (world_to_tm U0)) k2


(*
 *  S; F; W; E |− V                 // WAS in Aura
 *  S; F; W; E; V; U |− t : k        // New rule, rules out pathological
 *                                     types which don't upgrade/promote due to the
 *                                     presence of modal operators, such as let−as;
 *                                     for now, sticking with the old rule
 *  (c : t) \in S                  where c is a constructor
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−
 *  S; F; W; E; V; U |− c : t
 *
 *)
| wf_tm_ctr : forall W E c t V U,
      wf_worlds S F W E V U →
      sig_has_ctr c t S →
      wf_tm S F W E V U (tm_ctr c) t

(*
 *  S; F; W; E; V; U |− ok
 *  (c : t) \in S                   where c is an assumption
 * −−−−−−−−−−−−−−−−−−−−−−
 *  S; F; W; E; V; U |− c : t
 *
 *)
| wf_tm_assn : forall W E c t V U,
      wf_worlds S F W E V U →
      sig_has_assn c t S →
      wf_tm S F W E V U (tm_ctr c) t

(*
 *  S; F; W; E; V; U |− ok
 *  (x : t) @ V0 \in E
 *  V0 <= V
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 *  S; F; W; E; V; U |− x: t
 *)
| wf_tm_var : forall V0 W E x t V U,
       wf_worlds S F W E V U →
      binds_var x  t  V0 E→
      world_lteq  V0 V →
      wf_tm S F W E V U (tm_fv x) t

(*
 *  S; F;  W; E; bot;  bot|− u_1: k_1
 *  S; F;  W; E,x:u1 @bot ; V; U0 |− f^x:  u_2
 *  S; F;  E;  W; w_bot; w_bot |− (x:u_1)  −U0→ u2 : k
 *  k \in  {Type, Prop}
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 *  S; F;  W; E; V;  U |− abs f  :  (x:  u_1)  −U0> u_2
 *)
| wf_tm_abs: forall  L k  k1  W E V U u1 u2 f U0,
    wf_tm S F W E w_bot w_bot u1 k1 →
    wf_worlds S F W E V U0 →
    ( forall  x,
       x \notin  L →
       wf_tm S F W (E & x ~:  u1 @ w_bot) V U0 (f ^^  tm_fv  x)  (u2 ^^  tm_fv  x)  )  →
    wf_tm S F W E V U (tm_arr u1 u2 (world_to_tm U0)) k →
    k = K_TYPE \/ k = K_PROP →
    k1 = K_TYPE \/ k1 = K_PROP \/ u1 = K_TYPE \/ u1 = K_PROP →
    wf_tm S F W E V U (tm_abs u1 f (world_to_tm U0))
                            (tm_arr  u1 u2 (world_to_tm U0))



(*
 * S; F;  W; E; V;  U|− e1 : (x:t2)  −−U→ u
 * S; F;  W; E; V;  U2 |− e2 : t2
 * S; F;  W; E; V;  U |− u{e2/x}  : ku
 * S; F;  W; E; V;  U |− t2  :  k2
 *  value e2 /\  U2 = w_bot
 *       or
 *  lc  u /\  kinding   restrictions   /\  U2 = U
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * S; E |− e1 e2 :  u{e2/x}
*)
| wf_tm_app: forall  u ku k2 U2 W E V U e1 e2 t2 U_latent,
    wf_tm S F W E V U e1 (tm_arr t2 u (world_to_tm U_latent))
  → wf_tm S F W E w_bot U2 e2 t2
  → wf_tm S F W E V U t2 k2
  → wf_tm S F W E V U (u ^^ e2) ku
  → (value e2 /\  U2 = w_bot)
          \/
      (ku = K_TYPE /\ lc u /\  U2 = U)
          \/
      ((k2 = K_PROP \/ k2 = S_KIND) /\ lc u /\ U2 = U)
  → world_lteq  U_latent  U
  → wf_tm S F W E V U (tm_app e1 e2) (u ^^ e2)
(*
  e1 e2 is  ok when
```

```
   * e2 is a pure value
   * e1 is a non−dependent term, classified by a type, "an ml function,"
          that is, ku = K_TYPE
   * e1 is not dependent, e2 is a pure expression
        − and e2 is in prop, e2 : t2 : PROP
        − or e2 is type−level, e2 : t2 : KIND
*)

(*
 * S; F; W; E; V1; U |− e1 : t1
 * S; F; W; E, x:t1 @V1 ; V; U |− e2 :t
 * x \notin fv t
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * S; F; W; E; V; U |− let x at V1 = e1 in e2: t
 *)
| wf_tm_letat : forall L W E V U V1 e1 e2 t1 t k,
    wf_tm S F W E V1 U e1 t1 →
    wf_worlds S F W E V V1 →
    wf_tm S F W E w_bot w_bot t1 k →
    (k = K_TYPE \/ k = K_PROP) →
    ( forall x,
       x \notin L →
       wf_tm S F W (E & x ~: t1 @ V1) V U (e2 ^^ (tm_fv x)) t) →
    world_lteq V1 V →
    wf_tm S F W E V U (tm_letat (world_to_tm V1) e1 e2) t

(*
 * S; F; W; E; B; U |− e1 : t1 for B
 * S; F; W; E, x:t1 @B ; B; B |− e2 :t
 * x \notin fv t
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * S; F; W; E; V; U |− forbind x [t1 B] = e1 in e2: t for B
 *)
| wf_tm_forbnd: forall L W E V U B e1 e2 t1 t,
    wf_tm S F W E (w_prin B) U e1 (t1 FOR B) →
    wf_world S F W E (w_prin B) →
    wf_tm S F W E w_bot w_bot t1 K_TYPE →
    wf_tm S F W E w_bot w_bot (t FOR B) K_TYPE →
    wf_world S F W E V →
    ( forall x,
       x \notin L →
       wf_tm S F W (E & x ~: t1 @ (w_prin B)) (w_prin B) (w_prin B)
                      (e2 ^^ (tm_fv x)) (t FOR B)) →
    world_lteq (w_prin B) V →
    wf_tm S F W E V U (tm_forbnd (t FOR B) e1 e2)
                      (t FOR B)


(** Language−specific constants *)
```

```
(*
 * S;F;W;E;V;U |- v : T_PRIN
 * S;F;W;E;V;U|- P : K_PROP (value v)
 * --------------------------------
 * S;E |- v says p :  K_PROP
 *)
| wf_tm_P_SAYS: forall W E V U v P,
    wf_tm S F W E V U v T_PRIN
 → wf_tm S F W E V w_bot P K_PROP (* A shortcut: We could get this from
                                     higher in the derivation tree, but
                                     putting it here simplifies induction. *)
 → value v
 → wf_tm S F W E V U (tm_app (tm_app P_SAYS v) P) K_PROP

(* signatures:
 *
 * Signatures should be unambiguous, in standard Aure we require that
 * v and p check in the empty environment. Likewise, Aura-Conf requries
 * that the dynamic environment be empty as well.
 *
 * S;F;W;.;bot; U|- v : Prin
 * value v
 * S;F;W;.;bot; U |- p : Prop
 * S;F;W;E;V; U |- p : Prop
 * --------------------------
 * S;F;W;E;V; U |- sign(v, p) : a says p
 *)
| wf_tm_PF_SIGN : forall W E V U v p,
    wf_tm S F W empty w_bot w_bot v T_PRIN
 → value v
 → wf_tm S F W empty w_bot w_bot p K_PROP
 → wf_tm S F W E V U p K_PROP
 → wf_tm S F W E V U (tm_app (tm_app PF_SIGN v) p) (tm_app (tm_app P_SAYS v) p)

(*
 * S;F;W;E;V;U |- ok
 * ----------------------------
 * S;F;W;E;V;U |- Prin : Type
 *)
| wf_tm_T_PRIN : forall  W E V U,
    wf_worlds S F W E V U
 → wf_tm S F W E V U T_PRIN K_TYPE

 (*
  * S;F;W;E;V;U |- ok
  * S;F;W;E;V;U |- Bits: Type
  *)
| wf_tm_T_BITS : forall  W E V U,
    wf_worlds S F W E V U
 → wf_tm S F W E V U T_BITS K_TYPE
 (*
  * S;F;W;E;V;U |− t : Type
  * S;F;W;E;V;U |− A : Prin
  * −−−−−−−−−−−−−−−−−−−−−−−−−
  * S;F;W;E;V;U |− t for A : Type
  *)
| wf_tm_T_FOR : forall W E V U t a,
    wf_tm S F W E V w_bot t K_TYPE
 → wf_tm S F W E V w_bot a T_PRIN
 → wf_worlds S F W E V U
 → value t
 → value a
 → wf_tm S F W E V U (t FOR a) K_TYPE

(*
 * S;F;W;E;V;U |− P : K_PROP
 * −−−−−−−−−−−−−−−−−−−−−−−−−
 * S;F;W;E;V;U |− pf P : Type
 *)
| wf_tm_T_PF : forall W E V U P,
    wf_tm S F W E V w_bot P K_PROP (* another shortcut *)
 → wf_worlds S F W E V U
 → wf_tm S F W E V U (tm_app T_PF P) K_TYPE

(*
 * S;F;W;E;V;U |− p : P
 * S;F;W;E;V;U |− P : K_PROP
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * S;F;W;E;V;U |− return_pf p : (Pf P)
 *)
| wf_tm_TM_PF_RETURN : forall W E V U p P,
    wf_tm S F W E w_bot w_bot p P
 → wf_tm S F W E V U P K_PROP (* another shortcut *)
 → wf_worlds S F W E V U
 → wf_tm S F W E V U (tm_app TM_PF_RETURN p) (tm_app T_PF P)

(*
 * S;F;W;E;V;U |− e1 : (Pf P)
 * S;F;W;E;V;U |− e2 : (x:P) → (Pf Q)     x \notin (fv Q)
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * S;F;W;E;V;U |− bind e1 e2 : (Pf Q)
 *)
(* NOTE: x can't do a 'dependent−let' style typing rule
   because there's nothing appropriate to substitute for x in Q *)
| wf_tm_TM_PF_BIND : forall P W E V U e1 e2 Q,
    wf_tm S F W E V U e1 (tm_app T_PF P)
 → wf_tm S F W E V U e2 (tm_arr P (tm_app T_PF Q) STM_WORLD_BOT)
 → lc Q
 → wf_tm S F W E V U (tm_app (tm_app TM_PF_BIND e1) e2)
```

```
                          (tm_app T_PF Q)


(*
 *
 * S;F;A;E;bot;bot |- P : K_PROP
 * -----------------------------------------
 * S;F;A;E;V;A |- say P : (Pf (A says P))
 *)
| wf_tm_TM_SAY : forall W A E V U P ,
    wf_tm S F W E w_bot w_bot P K_PROP
 → wf_worlds S F W E V U
 → wf_tm S F W E w_bot w_bot A T_PRIN
 → world_lteq (w_prin A) U
 → value A
 → wf_tm S F W E V U
                          (SAY [A] P)
                          (tm_app T_PF (tm_app (tm_app P_SAYS A) P))

(*
 * S;F;W;E;w_bot;U|- v1 : T1    (value v1)
 * S;F;W;E;w_bot;U|- v2 : T1    (value v2)
 * S |- atomic [] T1
 * S;F;W; E,v1=v2 @ w_bot; V; U |- e1 : T
 * S;F;W;E;V;U|- e2 : T
 *-----------------------------------
 * S;F;W;E;V;U|- PIF v1 v2 e1 e2 : T
 *)
| wf_tm_TM_PIF : forall L W E V U v1 v2 e1 e2 T T1,
    wf_tm S F W E w_bot U v1 T1
 → wf_tm S F W E w_bot U v2 T1
 → atomic_type S T1
 → value v1
 → value v2
 → ( forall x,
      x \notin L →
      wf_tm S F W (E & x ~ (bnd_eq v1 v2 T1 w_bot)) V U e1 T
    )
 → wf_tm S F W E V U e2 T
 → wf_tm S F W E w_bot w_bot T1 K_TYPE
 → wf_tm S F W E V U
      (tm_app (tm_app (tm_app (tm_app TM_PIF v1) v2) e1) e2)
      T

(* Generalizing SELF rule *)
(*
 * S;F;W;E |- V
 * ----------------------
 * S;F;W;E;V |- self : Prin
 *)
```

```
| wf_tm_prin_const : forall W E V U p,
    wf_worlds S F W E V U
 → wf_tm S F W E V U (tm_prin p) T_PRIN

(*
 * S;F;W;E;V |− p : P
 * S;F;W;E;V |− P : K_PROP
 * S;F;W;E;V |− a : T_PRIN (value a)
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * S;F;W;E;V |− return_says a p : (a says P)
 *)
| wf_tm_PF_SAYS_RETURN : forall W E V U p P a,
    wf_tm S F W E V w_bot p P (* another shortcut *)
 → wf_tm S F W E V w_bot P K_PROP
 → wf_tm S F W E V U a T_PRIN
 → value a
 → wf_tm S F W E V U (tm_app (tm_app PF_SAYS_RETURN a) p)
                     (tm_app (tm_app P_SAYS a) P)


(*
 * S;F;W;E;V |− p1 : a says P
 * S;F;W;E;V |− p2 : (x:P) → a says Q   (x \notin (fv Q))
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * S;F;W;E;V |− bind p1 p2 : a says Q
 *)
| wf_tm_PF_SAYS_BIND : forall P W E V U a p1 p2 Q,
    wf_tm S F W E V w_bot p1 (tm_app (tm_app P_SAYS a) P)
 → wf_tm S F W E V w_bot p2 (tm_arr P (tm_app (tm_app P_SAYS a) Q) STM_WORLD_BOT)
 → lc Q
 → wf_worlds S F W E V U
 → wf_tm S F W E V U (tm_app (tm_app PF_SAYS_BIND p1) p2)
                     (tm_app (tm_app P_SAYS a) Q)

(*
 * Pattern Matching
 *)
|  wf_tm_mat_TYPE: forall W E V U e s branches result_typ ctr args k,
             wf_tm S F W E V U e s →
             wf_tm S F W E V U s K_TYPE →          (* s is a TYPE ... *)
              fully_applied_args s ctr args →      (*    ... and is a fully applied to (List.
                 reverse [args]) *)
             sig_has_ctr ctr k S →                 (*    ... which the sig says is a type con *)
              fully_applied_typ s ctr k →
             wf_tm S F W E V U result_typ K_TYPE → (* Homogeneous elimination only *)
             wf_branches S F W E V U s (List.rev args) branches result_typ →
                                                      (* check branches *)
             branches_cover S branches ctr →      (* Coverage check *)
             wf_tm S F W E V U (tm_mat e result_typ branches) result_typ
```

```
| wf_tm_mat_PROP: forall W E V U e s branches result_typ ctr args k,
                 wf_tm S F W E V U e s →
                 wf_tm S F W E V U s K_PROP →           (* s is a PROP ... *)
                 fully_applied_args s ctr args →        (*   ... and is a fully applied to (List.
                     reverse [args]) *)
                 sig_has_ctr ctr k S →                  (*   ... which the sig says is a type con *)
                 fully_applied_typ s ctr k →
                 wf_tm S F W E V U result_typ K_PROP → (* Homogeneous elimination only *)
                 wf_branches S F W E V U s (List.rev args) branches result_typ →
                                                          (* check branches *)
                 branches_cover S branches ctr →       (* Coverage check *)
                 wf_tm S F W E V U (tm_mat e result_typ branches) result_typ

(*  S; W; F; E; V; U; |− ok
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 *  S; W; F; E; V; U; |− enc(A,e,n): bits  *)
| wf_tm_enc: forall W E V U A e n,
    wf_worlds S F W E V U →
    wf_tm S F W E V U (tm_enc A e n) T_BITS

(*  S; W; F; E; V; U |− e: t
 *  S; W; F; E; V; U |− ok
 *  S; W; F; E; V; U |− t FOR A : TYPE
 *  A <= V
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 *  S; W; E; V; U; |− return (t for A) e: (t FOR A) *)
| wf_tm_TM_FOR_RET: forall W E V U A e t,
    wf_tm S F W E (w_prin A) (w_prin A) e t →
    wf_worlds S F W E V U →
    wf_tm S F W E w_bot w_bot (t FOR A) K_TYPE →
    world_lteq (w_prin A) V →
    wf_tm S F W E V U (FORRET [A, t] e ) (t FOR A)

(*
 * S;F;W;E;V;U |− ok
 * good−enough E e (t FOR A)
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * S;F;W;E;V;U |− ASBITS e : bits
 *)
| wf_tm_TM_AS_BITS: forall W E V U e t A,
    wf_worlds S F W E V U →
    ge_tm E e (t FOR A) →
    wf_tm S F W E V U (ASBITS e) (T_BITS)

| wf_tm_P_ISA: forall W E V U e t B,
    wf_tm S F W E V U (t FOR B) K_TYPE →
    wf_tm S F W E V U e T_BITS →
    value e →
    wf_tm S F W E V U (e ISA (t FOR B)) K_PROP
```

```
(*
 * S;F;W;E;V;U |− e : t FOR A
 * A <= V
 * A <= U
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * S;F;W;E;V;U |− run e : t
 *)
| wf_tm_TM_FOR_RUN: forall W E V U e t A,
    wf_tm S F W E V U e (t FOR A) →
    world_lteq (w_prin A) V →
    world_lteq (w_prin A) U →
    wf_tm S F W E V U (FORRUN e) t

(* Conversion rule:
 *
 * S;F;W;E;V;U |− e : t1
 * S;F;W;E;V;U |− t1 : K_TYPE
 * S;F;W;E;V;U |− t2 : K_TYPE
 * S;E |− t1 == t2
 * −−−−−−−−−−−−−−−−−−−
 * S;E |− cast(e, t2, CONV) : t2
 *)
| wf_tm_cast_conv : forall W E V U e t1 t2,
    wf_tm S F W E V U e t1
 → wf_tm S F W E V U t2 K_TYPE (* added to make proof easier *)
 → converts E V t1 t2
 → wf_tm S F W E V U (tm_cast e t2 STM_CONV_CAST) t2

(* Fact Cast *)
(*
 * Enc(a,e,n): t for B \in F
 * S;F;W;E;V;U |− t for B : TYPE
 * B <= V
 * −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * S;F;W;E;V;U |− cast( Enc(a, e, n), t for B, TRUE ),
 *)
| wf_tm_cast_fact : forall W E V U a e n t B,
    fctx_has F (tm_enc a e n) (t FOR B) →
    wf_tm S F W empty w_bot w_bot (t FOR B) K_TYPE →
      (* Facts with free type variables don't make sense *)
    wf_worlds S F W E V U →
    world_lteq (w_prin B) V →
    wf_tm S F W E V U (tm_cast (tm_enc a e n) (t FOR B) (STM_TRUE_CAST))
                      (t FOR B)

(* Decryption Cast *)
(*
 * S;F;W; . ;B;B |− e : t
 * S;F;W;E;V;U |− t for B : Type
 * B <= V
 * ---------------------------------------------------------
 * S;F;B;E;V;U |- cast( Enc(B, e, n), t for B, TRUE ),
 *)
| wf_tm_cast_dec: forall E W V U e n t B,
    wf_tm S F W empty (w_prin B) (w_prin B) e t →
        (* e is not allowed to dynamically reference variables (this
             would break the strengthening lemma, and also does not
             make sense. *)
    wf_tm S F W empty w_bot w_bot (t FOR B) K_TYPE →
    wf_worlds S F W E V U →
    wf_world S F W empty (w_prin B) →
    world_lteq (w_prin B) V →
    world_lteq (w_prin B) W →
    wf_tm S F W E V U
                       (tm_cast (tm_enc B e n) (t FOR B) (STM_TRUE_CAST))
                       (t FOR B)

(* Justified Cast *)
(*
 *   S; F; W; E; V; U |- e2 : pf (A says (e1 isa (t for B)))
 *   S; F; W; E; V; U |- e2 : e1: bits
 *   S; F; W; E; _|_; _|_ |- t for B: TYPE
 *   ------------------------
 *)
| wf_tm_cast_just: forall W E V U e1 t A B e2,
    wf_tm S F W E V U e2 (PF (A SAYS (e1 ISA (t FOR B)))) →
    wf_tm S F W E V U e1 T_BITS →
    wf_tm S F W E w_bot w_bot (t FOR B) K_TYPE →
    value e1 →
    wf_tm S F W E V U (tm_cast e1 (t FOR B) (tm_mapp STM_JUST_CAST e2))
                       (t FOR B)

(* need secondary judgment to type check branches that records the
     type we are matching *)
(*   wf_branches S E branches tctr args result_typ *)
with wf_branches (S:sig) (F: fact_ctx ):
  world → env → world → world → tm → list tm → tm → tm → Prop :=
| wf_branches_None : forall W E V U s args result_type,
        wf_branches S F W E V U s args (TM_NONE) result_type

| wf_branches_Some : forall W E V U s args ectr ectr_typ body body_type rest result_type ,
        wf_branches S F W E V U s args rest result_type →
        sig_has_ctr ectr ectr_typ S →
        wf_tm S F W E V U body body_type →
        wf_brn S F s args ectr_typ body_type result_type U →
        wf_branches S F W E V U s args (tm_brn ectr body rest) result_type

(* wf_brn   *)
with wf_brn (S:sig) (F: fact_ctx ):
        tm → ( list tm) → tm → tm → tm → world → Prop :=
  (* Base case:
       – The constructor takes no arguments, the datatype has no remaining parameters
         so the type of the body must just be the result type of the match expression
       – Example: when checking the "Z" case of the datatype Nat:

         data Nat : Type {
         | Z : Nat
         | S : Nat → Nat
         }

         when pattern matching against a Nat, we have:
           match e : Nat {
           | Z → body_Z
           | S → \ (n:Nat).body_S
           }
   *)
| wf_brn_base :
     forall s result_type (U: world),
     wf_brn S F s nil s result_type result_type U
  (* Inductive case:
       – The constructor takes arguments in addition to the datatype parameters
       – The parameters have already been stripped off, so the type of the body
         must take an argument
       – Continuing the example, when checking the successor branch for type Nat:
         the constructor S has type Nat → Nat so the body should have type
         Nat → result_typ
       – The body should be of the form \ ([x]:Nat).body
       – Note: This rule does not allow the result typ to depend on the
         constructor argument
       – Note: latent effects on branches must be bounded by provided world
  *)
| wf_brn_constructor_arg :
     forall L s T1 T U result_typ (eff_here: world) (eff_bound: world),
     (forall x,
        x \notin L →
        x \notin (fv result_typ) →
         wf_brn S F s nil (T^^(tm_fv x)) (U^^(tm_fv x)) result_typ eff_bound) →
     world_lteq eff_here eff_bound →
     wf_brn S F s nil (tm_arr T1 T (world_to_tm eff_here))
                      (tm_arr T1 U (world_to_tm eff_here)) result_typ eff_bound

  (* Inductive case:
     – The datatype has a parameter (e.g. List : Type → Type)
     – The type of the scrutinee is thus [tctr] applied to [args]
     – We witness the generated equality by opening up the types
       using the arguments in [args]

     – For example: when checking the 'cons' case of a match against
       an expression of type (List Nat) we have [tctr] = List and
       [args] = [Nat] The branch should look like:
         match e {
         |  nil  → body_nil
         |  cons → \  (hd:0).\  ( tl : List  1). body_cons
         }
  *)
| wf_brn_datatype_param:
     forall  s arg args T1 T U eff_here  result_type  eff_bound,
       wf_brn S F s args (T ˆˆ  arg)  U result_type  eff_bound →
       world_lteq  eff_here  eff_bound →
       wf_brn S F s  (arg :: args)
                       (tm_arr  T1 T (world_to_tm  eff_here ) )  U
                       result_type  eff_bound
.
```

C.11 Signature Well Formedness

```
(* wf_sig: Ensures that a list of (mutually recursive sets of) data
 *         type declarations is well formed.
 *
 * |- S wf
 *
 *)

Inductive wf_dom_defns: defns →Prop:=
 | wf_dom_defns_nil: wf_dom_defns nil
 | wf_dom_defns_cons: forall c k cdl d,
                          wf_dom_defns d →
                           disjoint (dom_ctr_decls cdl) (dom_defns d) →
                          ~In c (dom_ctr_decls cdl) →
                          ~In c (dom_defns d) →
                          wf_dom_defns ((c,k,Some cdl)::d)
 | wf_dom_defns_cons_none: forall c k d,
                          wf_dom_defns d →
                          ~In c (dom_defns d) →
                          wf_dom_defns ((c,k,None)::d).


Inductive wf_dom_sig: sig → Prop:=
 | wf_dom_sig_nil: wf_dom_sig nil
 | wf_dom_sig_cons_assn: forall c k S,
                        wf_dom_sig S →
                        ~ In c (dom_sig S) →
                        wf_dom_sig (bundle_assert c k::S)
 | wf_dom_sig_cons_defns: forall d S,
                        wf_dom_defns d →
                        wf_dom_sig S →
                         disjoint (dom_defns d) (dom_sig S) →
                        wf_dom_sig (bundle_defns d::S).

Inductive wf_sig : sig → Prop :=

(*
  ----------------
  |- nil wf_sig
*)
| wf_sig_nil : wf_sig nil


(* assertion *)
| wf_assertion : forall c k S,
                   wf_sig S
                 → wf_tm S fc_empty w_bot empty w_bot w_bot k S_KIND
                 → constructs k K_PROP
                 → ~(In c (dom_sig S))
```

```
                    → wf_sig ((bundle_assert c k) :: S)

(* Note: [b] means the bundle derived from b by considering only the type constructors in b
   (and ignoring the constructors associated with them).

 |- S wf_sig
 S |- b :: Type wf_bundle_tctrs
 nil ; S,[b] |- b wf_bundle_ctrs
 --------------------------------
  |- S,b wf_sig
*)

| wf_sig_cons_Type : forall (d:defns) (S:sig),
                        wf_sig S
                     → wf_bundle_tctrs S d K_TYPE
                     → disjoint (dom_defns d) (dom_sig S)
                     → wf_bundle_ctrs nil ((bundle_defns (get_tctr_defns_of d)) :: S) d
                     → wf_dom_defns d
                     → wf_sig ((bundle_defns d)::S)


(* Note: [b] means the bundle derived from b by considering only the type constructors in b
   (and ignoring the constructors associated with them).

 |- S wf_sig
 S |- b :: Prop wf_bundle_tctrs
 [b] ; S,[b] |- b wf_bundle_ctrs
 --------------------------------
  |- S,b wf_sig
*)
| wf_sig_cons_Prop : forall (d:defns) (S:sig),
                        wf_sig S
                     → wf_bundle_tctrs S d K_PROP
                     → disjoint (dom_defns d) (dom_sig S)
                     → wf_bundle_ctrs (dom_defns (get_tctr_defns_of d)) ((bundle_defns (get_tctr_defns_of
                          d)) :: S) d
                     → wf_dom_defns d
                     → wf_sig ((bundle_defns d)::S)


(* Ensures that each type constructor in the mutually recursive datatype
   constructs an object (Type or Prop) of the same kind [k]
   This means that props and types can't be mutually recursively defined.
*)
with wf_bundle_tctrs : sig → defns → tm → Prop :=

(*
  S;. |- k :: SORT
  ------------------------
  S |- nil :: k wf_bundle_tctrs
*)
| wf_bundle_tctrs_nil    : forall S k,
                           wf_tm S fc_empty w_bot empty w_bot w_bot k S_KIND
                             (* Requires that k = Type or Prop *)
                         → wf_bundle_tctrs S nil k

(*
   S |- b :: k wf_bundle_tctrs
   c \notin dom (S,b)
   S;. |- k : KIND
   k == (x1:t1) → (x2:t2) → ... → K
   --------------------------------------
   S |- (c,k,cdl) :: K wf_bundle_tctrs

*)
| wf_bundle_tctrs_cons : forall S c k K cdl d,
                           wf_bundle_tctrs S d K
                         → ~(In c (dom_defns d)) (* c not already defined *)
                         → wf_tm S fc_empty w_bot empty w_bot w_bot k S_KIND
                         → constructs k K
                         → wf_bundle_tctrs S ((c, k, cdl) :: d) K

(* First argument is a list of type constructors that must appear
   positively in expression constructor types *)
with wf_bundle_ctrs : ( list ctr ) → sig → defns → Prop :=

(* Note: could add |- S wf_sig
     (but that does a lot of re-checking of the signature)

   ------------------------------
   P ; S |- nil wf_bundle_ctrs
*)
| wf_bundle_ctrs_nil    : forall P S,
                            wf_bundle_ctrs P S nil


(*
  P ; S |- b wf_bundle_ctrs
  P ; S |- c : k {cdl} wf_ctr_decls
  dom (ctrs) \cap dom (b) = empty
  dom (ctrs) \cap dom S = empty
  ------------------------------------
  P ; S |- b and c:t {cdl} wf_bundle_ctrs

*)
| wf_bundle_ctrs_cons : forall P S c k cdl d,
                            wf_bundle_ctrs P S d
                            (* cdl is a well-formed constructor
                             * declaration list for c *)
                          → wf_ctr_decls P S c k cdl
                          → wf_bundle_ctrs P S ((c, k, Some cdl)::d)
```

```
| wf_bundle_ctrs_cons_none : forall  P S c k d,
                             wf_bundle_ctrs  P S d
                           → wf_bundle_ctrs  P S ((c,  k,  None)::d)



with wf_ctr_decls  :  ( list   ctr )  → sig  →  ctr  → tm → ctr_decls  → Prop :=
(*
  ––––––––––––––––––
  P ; S |− c : k {} wf_ctr_decls
*)
|  wf_ctr_decls_nil    :  forall  P S c k,
                       wf_ctr_decls  S P c k  nil

(*
  P ; S |− c : k {cdl} wf_ctr_decls
  ctr \notin dom(cdl)
  S ; . |− t : K
  t == (x1:t1) → (x2:t2) →  ...  → (c x1  ...  xm)
  c_inst = (c x1  ...  xm)  is saturated at kind k
  constructors in P appear strictly  positively  in  t
  ––––––––––––––––––––––––––––––––––––––––––––––––––––
  P ; S |−
*)
| wf_ctr_decls_cons  :  forall  K args c_inst  P S c k  ctr  n m t cdl,
                        wf_ctr_decls  P S c k cdl
                     → ~( In  ( ctr ,  n)  (dom_ctr_decls cdl))   (* Constructor is used only once *)
                     → wf_tm S fc_empty w_bot empty w_bot w_bot t K (* Make sure the type definition
                          is of the right kind *)
                     → has_level  t  m
                     → constructs  t  c_inst                (*  t constructs an instance c_inst  *)
                     →  fully_applied_args  c_inst  c args    (* that is fully applied [c] of kind [k] *)
                     →  fully_applied_typ   c_inst  c k
                     → n = List.length args                 (*  n is the number of args *)
                     → ~(m < n)
                     → all_tm_bv  (m−n) args                (*  and all the arguments are bound variables *)
                     → positive  P t                        (* Ensure tycons in P only occur positively  *)
                     →  wf_ctr_decls  P S c k  ((( ctr ,  n),  t ) :: cdl)
.
```

C.12 Step Relation

```
Inductive step_matches_ctr: tm → ctr → tm → (tm ∗ nat ∗ list tm ∗ list tm) → Prop :=
| step_matches_ctr_base: forall c body n,
       step_matches_ctr (tm_ctr (c, n)) (c, n) body (body, n, nil , nil )

| step_matches_ctr_strip_type_arg : forall v1 v2 m c body args,
       value v2
    → step_matches_ctr v1 c body (body, m, args, nil )
    → m > 0
    → step_matches_ctr (tm_app v1 v2) c body (body, m−1, args++(v2::nil), nil )

| step_matches_ctr_rec: forall v1 v2 c body e args args1,
       value v2
    → step_matches_ctr v1 c body (e, 0, args, args1)
    → step_matches_ctr (tm_app v1 v2) c body ((tm_app e v2), 0, args, args1++ v2::nil)
.

Inductive step_matches_branches : tm →tm →tm →Prop :=
| step_matches_here: forall v c body rest e args args1,
       step_matches_ctr v c body (e, 0, args, args1)
    → step_matches_branches v (tm_brn c body rest) e

| step_matches_earlier: forall v c body rest e,
       step_matches_branches v rest e
    → ~ (exists r, step_matches_ctr v c body r)
    → step_matches_branches v (tm_brn c body rest) e
.

Inductive step_config : Type :=
  sc_basic : tm → nat → step_config.

Notation "{| t , n |}" := (sc_basic t n).

Inductive step (S: sig) (W:world) ( F_init : fact_ctx ) :
  step_config → step_config → fact_ctx → Prop :=
(∗ Computational rules ∗)

| step_app: forall n t e v U,
       value v
    → step S W F_init {|(tm_app (tm_abs t e U) v), n|}
                      {|(e ^^ v), n |}
                      fc_empty

| step_letat : forall n V v1 e2,
       value v1
    → step S W F_init {| tm_letat V v1 e2, n|} {|e2 ^^ v1, n|} fc_empty

| step_forbnd: forall n A t v e,
       value v
    → step S W F_init
               {|FORBND [(tm_prin A), t, v, e], n|}
               {|tm_cast (tm_enc (tm_prin A)
                             ( tm_letat (tm_mapp
                                          STM_WORLD_PRIN
                                          (tm_prin A)) (FORRUN v)
                                                       (FORRUN e))
                             n)
                     ( t FOR (tm_prin A))
                     STM_TRUE_CAST, 1+n |}
               (fc_cons (tm_enc (tm_prin A)
                               ( tm_letat (tm_mapp
                                            STM_WORLD_PRIN
                                            (tm_prin A)) (FORRUN v)
                                                         (FORRUN e))
                               n)
                      ( t FOR (tm_prin A)) fc_empty)

| step_pf_bind: forall n p f,
      value f
    → value p
    → step S W F_init
               {|tm_app (tm_app TM_PF_BIND (tm_app TM_PF_RETURN p)) f, n|}
               {|tm_app f p, n|} fc_empty

| step_say: forall A n p,
      world_lteq (w_prin A) W →
      step S W F_init
               {|SAY [ A ] p, n|}
               {|tm_app TM_PF_RETURN (tm_app (tm_app PF_SIGN A) p), n|}
               fc_empty

| step_pif1 : forall n v1 v2 e1 e2,
      v1 = v2 (* note -- in principle, this should be equality defined
                     for runtime-principal values *)
    → step S W F_init
           {|tm_app (tm_app (tm_app (tm_app TM_PIF v1) v2) e1) e2, n|}
           {|e1, n|} fc_empty

| step_pif2 : forall n v1 v2 e1 e2,
      v1 <> v2 (* note -- in principle, this should be equality defined
                      for runtime-principal values *)
    → step S W F_init
         {|tm_app (tm_app (tm_app (tm_app TM_PIF v1) v2) e1) e2, n|}
         {|e2, n|} fc_empty

| step_mat: forall n v t branches e,
      value v
    → step_matches_branches v branches e
    → step S W F_init {|tm_mat v t branches, n|} {|e, n|} fc_empty
```

```
| step_ret_for : forall A e t n,
      step S W F_init
                  {|FORRET [(tm_prin A), t] e , n|}
                  {|tm_cast (tm_enc (tm_prin A) e n)
                                       (t FOR (tm_prin A)) STM_TRUE_CAST, 1+n |}
                  (fc_cons (tm_enc (tm_prin A) e n) (t FOR (tm_prin A)) fc_empty)

| step_as_bits : forall e1 e2 t n0 reason n,
    value (tm_cast (tm_enc e1 e2 n0) t reason) →
    step S W F_init
               {|ASBITS (tm_cast (tm_enc e1 e2 n0) t reason), n|}
               {|tm_enc e1 e2 n0, n|} fc_empty

| step_for_run_ok : forall n e0 A0 n0 t r,
     value (tm_cast (tm_enc A0 e0 n0)
                    (t FOR A0) r)
   → wf_tm S F_init W empty (w_prin A0) (w_prin A0) e0 t
   → world_lteq (w_prin A0) W
   → step S W F_init
                  {|FORRUN (tm_cast (tm_enc A0 e0 n0)
                                       (t FOR A0) r), n |}
                  {|e0, n|} fc_empty

| step_for_run_ill_typed : forall n e0 A0 n0 t p,
     value (tm_cast (tm_enc A0 e0 n0)
                            (t FOR A0)
                            (tm_mapp STM_JUST_CAST p))
   → world_lteq (w_prin A0) W (* can decrypt *)
   → ~ (wf_tm S F_init W empty (w_prin A0) (w_prin A0) e0 t) (* but can't typecheck *)
   → step S W F_init
                  {|FORRUN (tm_cast (tm_enc A0 e0 n0)
                                       (t FOR A0)
                                       (tm_mapp STM_JUST_CAST p)), n |}
                  {|FAIL p, n|} fc_empty

| step_for_run_junk : forall n (A0: tm) e0 n0 t B p,
     value (tm_cast (tm_enc A0 e0 n0)
                    (t FOR B)
                    (tm_mapp STM_JUST_CAST p))
   → world_lteq (w_prin B) W (* decryption ought to succeed *)
   → A0 <> B (* but the cyphertext isn't encrypted with the correct key,
                or maybe not any valid key at all*)
   → step S W F_init
                  {|FORRUN (tm_cast (tm_enc A0 e0 n0)
                                       (t FOR B)
                                       (tm_mapp STM_JUST_CAST p)), n |}
                  {|FAIL p, n|} fc_empty
```

```
| step_cast: forall n v t,
      value v
   → step S W F_init {|tm_cast v t STM_CONV_CAST, n|}
                     {|v, n|} fc_empty

(* Stuctural congruence rules *)

| step_app_cong1: forall n e1 e1' e2 n' F,
      step S W F_init {|e1, n|} {|e1', n'|} F
   → step S W F_init {|tm_app e1 e2, n|} {|tm_app e1' e2, n'|} F

| step_app_cong2: forall n t e1 U e2 e2' n' F,
      step S W F_init {|e2, n|} {|e2', n'|} F
   → step S W F_init {|tm_app (tm_abs t e1 U) e2, n|}
                     {|tm_app (tm_abs t e1 U) e2', n'|} F

| step_app_cong3: forall n v e2 e2' n' F,
      applied_value v
   → step S W F_init {|e2, n|} {|e2', n'|} F
   → step S W F_init {|tm_app v e2, n|} {|tm_app v e2', n'|} F

| step_letat_cong1: forall n V e1 e2 e1' n' F,
      step S W F_init {|e1, n|} {|e1',n'|} F
   → step S W F_init {|tm_letat V e1 e2, n|} {|tm_letat V e1' e2, n'|} F

| step_forbnd_cong: forall n V e1 e2 e1' n' F,
      step S W F_init {|e1, n|} {|e1',n'|} F
   → step S W F_init {|tm_forbnd V e1 e2, n|} {|tm_forbnd V e1' e2, n'|} F

| step_pf_ret_cong: forall n e1 e1' n' F,
     step S W F_init {|e1, n|} {|e1', n'|} F
 →  step S W F_init {|tm_app TM_PF_RETURN e1, n|}
           {|tm_app TM_PF_RETURN e1', n'|} F

| step_pf_bind_cong1: forall n e1 e1' e2 n' F,
      step S W F_init {|e1, n|} {|e1', n'|} F
   → step S W F_init {|tm_app (tm_app TM_PF_BIND e1) e2, n|}
                     {|tm_app (tm_app TM_PF_BIND e1') e2, n'|} F

| step_pf_bind_cong2: forall n e1 e2 e2' n' F,
      value e1
   → step S W F_init {|e2, n|} {|e2', n'|} F
   → step S W F_init {|tm_app (tm_app TM_PF_BIND e1) e2, n|}
                     {|tm_app (tm_app TM_PF_BIND e1) e2', n'|} F

| step_for_run_cong: forall n e e' n' F,
      step S W F_init {|e, n|} {|e', n'|} F
   → step S W F_init {|FORRUN e, n|} {|FORRUN e', n'|} F

| step_mat_cong: forall n e1 e1' branches t n' F,
      step S W F_init {|e1, n|} {|e1', n'|} F
   → step S W F_init {|tm_mat e1 t branches, n|}
            {|tm_mat e1' t branches, n'|} F

| step_cast_cong1: forall n e1 e1' t reason n' F,
      step S W F_init {|e1, n|} {|e1', n'|} F
   → step S W F_init {|tm_cast e1 t reason, n|}
            {|tm_cast e1' t reason, n'|} F

| step_cast_cong2: forall n e1 e2 e2' t n' F,
      value e1
   → step S W F_init {|e2, n|} {|e2', n'|} F
   → step S W F_init {|tm_cast e1 t (tm_mapp STM_JUST_CAST e2), n|}
            {|tm_cast e1 t (tm_mapp STM_JUST_CAST e2'), n'|} F
.
```

C.13 Blame

```
Inductive blame_mapp_op: tm →Prop :=
  bmo_for: blame_mapp_op T_FOR
| bmo_isa: blame_mapp_op P_ISA
| bmo_forret:  blame_mapp_op TM_FOR_RET
| bmo_say:     blame_mapp_op TM_SAY
| bmo_forrun: blame_mapp_op TM_FOR_RUN
| bmo_justcast: blame_mapp_op STM_JUST_CAST
| bmo_ind:      forall  e1 e2, blame_mapp_op e1 →blame_mapp_op (tm_mapp e1 e2).

(∗ This  definition  is  is  simplified  by making all  builtin
         operations constants ∗)
Inductive blames (p: tm): tm → Prop :=
  bl_exact  :  blames p (FAIL p)
| bl_app_l  :  forall  e1 e2,
                  blames p e1 →blames p (tm_app e1 e2)
| bl_app_r  :  forall  e1 e2,
                  blames p e2 →blames p (tm_app e1 e2)
| bl_mapp_l :  forall  e1 e2,
                  blames p e1 →blames p (tm_mapp e1 e2)
| bl_mapp_r :  forall  e1 e2,
                  blame_mapp_op e1 →
                  blames p e2 →blames p (tm_mapp e1 e2)
|  bl_letat   :  forall  U e1 e2,
                  blames p e1 →blames p (tm_letat  U e1 e2)
| bl_forbnd  :  forall  U e1 e2,
                  blames p e1 →blames p (tm_forbnd U e1 e2)
| bl_mat  :  forall  e  t  bs,
                  blames p e →blames p (tm_mat e t bs)
| bl_cast_1  :  forall  e  t  r ,
                  blames p e →blames p (tm_cast e  t  r )
| bl_cast_2  :  forall  e  t  r ,
                  blames p r → blames p (tm_cast e  t  r ) .
```

C.14 Similarity

```
Inductive similar (W: world): tm → tm → Prop :=
| sim_fv:   forall x, similar W (tm_fv x) (tm_fv x)
| sim_bv:   forall n, similar W (tm_bv n) (tm_bv n)
| sim_con: forall n, similar W (tm_con n) (tm_con n)
| sim_ctr: forall c, similar W (tm_ctr c) (tm_ctr c)
| sim_app: forall t1 t2 u1 u2,
                 similar W t1 t2 →
                 similar W u1 u2 →
                 similar W (tm_app t1 u1) (tm_app t2 u2)
| sim_mapp: forall t1 t2 u1 u2,
                 similar W t1 t2 →
                 similar W u1 u2 →
                 similar W (tm_mapp t1 u1) (tm_mapp t2 u2)
| sim_prin: forall p, similar W (tm_prin p) (tm_prin p)
| sim_abs: forall t1 e1 U1 t2 e2 U2,
                similar W t1 t2 →
                similar W e1 e2 →
                similar W U1 U2 →
                similar W (tm_abs t1 e1 U1) (tm_abs t2 e2 U2)
| sim_arr: forall s1 t1 U1 s2 t2 U2,
                similar W s1 s2 →
                similar W t1 t2 →
                similar W U1 U2 →
                similar W (tm_arr s1 t1 U1) (tm_arr s2 t2 U2)
| sim_mat: forall e1 t1 b1 e2 t2 b2,
                similar W e1 e2 →
                similar W t1 t2 →
                similar W b1 b2 →
                similar W (tm_mat e1 t1 b1) (tm_mat e2 t2 b2)
| sim_brn: forall c e1 e2 b1 b2,
               similar W e1 e2 →
               similar W b1 b2 →
               similar W (tm_brn c e1 b1) (tm_brn c e2 b2)
| sim_cast: forall e1 t1 r1 e2 t2 r2,
                similar W e1 e2 →
                similar W t1 t2 →
                similar W r1 r2 →
                similar W (tm_cast e1 t1 r1) (tm_cast e2 t2 r2)
| sim_enc_decrypt: forall a e1 e2 n1 n2,
                world_lteq (w_prin (tm_prin a)) W →
                similar W e1 e2 →
                similar W (tm_enc (tm_prin a) e1 n1)
                          (tm_enc (tm_prin a) e2 n2)
| sim_enc_opaque: forall a1 a2 e1 e2 n1 n2,
                ~(world_lteq (w_prin (tm_prin a1)) W) →
                ~(world_lteq (w_prin (tm_prin a2)) W) →
                similar W (tm_enc (tm_prin a1) e1 n1)
                          (tm_enc (tm_prin a2) e2 n2)
```

```
| sim_letat : forall U1 s1 t1 U2 s2 t2,
                 similar W U1 U2 →
                 similar W s1 s2 →
                 similar W t1 t2 →
                 similar W (tm_letat s1 t1 U1) (tm_letat s2 t2 U2)
| sim_forbnd: forall U1 s1 t1 U2 s2 t2,
                 similar W U1 U2 →
                 similar W s1 s2 →
                 similar W t1 t2 →
                 similar W (tm_forbnd s1 t1 U1) (tm_forbnd s2 t2 U2).
```