A Semantics for Attestation Protocols using Session Types in Coq

Adam Petz

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science.

Thesis Committee:

Dr. Perry Alexander: Chairperson

Dr. Andy Gill

Dr. Prasad Kulkarni

Date Defended

The Thesis Committee for Adam Petz certifies
that this is the approved version of the following thesis:

A Semantics for Attestation Protocols using Session Types in Coq

Committee:

Chairperson

Date Approved

Acknowledgements

I have wonderful friends and family, and I am forever grateful for their undying support of my pursuits, academic and otherwise. I thank my advisor, Dr. Perry Alexander, for his continued generousity of time, knowledge, experience, and energy. During these past two years, I have grown as a student, researcher, and human being under his patient guidance. To all of the EECS faculty and staff at KU, including at ITTC, thank you for freely sharing your knowledge and passion. I'd especially like to thank the rest of my thesis committee, Dr. Andy Gill and Dr. Prasad Kulkarni, for their support both in the classroom and in the lab. Last but not least, thanks to my fellow students, especially my labmates. I value your friendship, passion, empathy, and all of the ping pong breaks.

Abstract

As our world becomes more connected, the average person must place more trust in cloud systems for everyday transactions. We rely on banks and credit card services to protect our money, hospitals to conceal and selectively disclose sensitive health information, and government agencies to protect our identity and uphold national security interests. However, establishing trust in remote systems is not a trivial task, especially in the diverse, distributed ecosystem of todays networked computers. *remote attestation* is a mechanism for establishing trust in a remotely running system where an *appraiser* requests information from a target that can be used to evaluate its operational state. The target responds with evidence providing configuration information, run-time measurements, and authenticity meta-evidence used by the appraiser to determine if it trusts the target system. For remote attestation to be applied broadly, we must have *attestation protocols* that perform operations on a collection of applications, each of which must be measured differently. Verifying that these protocols behave as expected and accomplish their diverse attestation goals is a unique challenge. An important first step is to understand the structural properties and execution patterns they share. In this thesis I present a semantic framework for attestation protocol execution within the Coq verification environment including a protocol representation based on Session Types, a dependently typed model of perfect cryptography, and an operational execution semantics. The expressive power of dependent types constrains the structure of protocols and supports precise *claims* about their behavior. If we view attestation protocols as programming language expressions, we can borrow from standard language semantics techniques to model their execution. The proof framework ensures desirable properties of protocol execution that hold for *all* protocols. Within this framework, it is feasible to state and prove specialized properties such as authenticity and secrecy for individual protocols.

Contents

Chapter 1

Introduction

As our world becomes more connected, the average person must place more trust in remote (cloud) systems for everyday transactions. We rely on banks and credit card services to protect our money, hospitals to conceal and selectively disclose sensitive health information, and government agencies to protect our identity and uphold national security interests. However, establishing trust in remote systems is not a trivial task, especially in the diverse, distributed ecosystem of todays networked computers.

In their foundational paper, Haldar et al. [2004] describe *remote attestation* as a mechanism for establishing trust in a remotely running system. In remote attestation, an *appraiser* requests information from a *target* that can be used to evaluate its operational state. The target responds with *evidence* providing configuration information, run-time measurements and authenticity meta-evidence used by the appraiser to determine if it trusts the target system. For remote attestation to be applied broadly, we must have *attestation protocols* [Coker et al., 2011] that perform operations on a collection of applications, each of which must be measured differently. Veryfing that these protocols behave as expected and accomplish their diverse attestation goals is a unique challange. An important first step is to understand the structural properties and execution patterns they share.

In this thesis we present a semantic framework for attestation protocol execution within the Coq verification environment [Coq development team, 2016]. Our framework includes a protocol representation, a dependently typed model of perfect cryptography, and an operational execution semantics. We leverage the expressive power of dependent types to constrain the structure of protcols and also to make precise *claims* about their execution behavior. Armed with this definitional framework, we build a proof framework that ensures desirable properties of protocol execution, such as progress and termination, that hold for *all* protocols. The entire framework, including all Coq definitions, functions, and proofs from this thesis are available in an online Github repository [Petz, 2016].

Session Types [Honda et al., 1998, Pucella and Tov, 2008] are a type system that describes communication protocols. In this thesis, we leverage the use of Session Types to ensure valid construction of protocols and to guide proofs about their execution semantics. Not only do Session Types reflect the overall structure of the protocol (its flow of communication and branching events), they are also parameterized by descriptive message types that provide a detailed description of the structure of messages being transmitted. These messages are orthogonal to the Session Type framework, and are implemented as a dependently typed model of perfect cryptography. They capture the often complex cryptographic structure of the evidence packages built and shared by attestation protocols.

Although the Session Types only describe message types at explicit communication events, they implicitly prevent certain “bad” local protocol operations statically. For example, they prevent decryption of a Basic message (that is not encrypted) and using a Basic message where a Key is expected. Session Types also support protocol branching where both participants in the protocol are prepared to branch and remain compatible. One member *offers* two protocol branches, while the other *chooses* a branch and notifies the other member. This branching capability may be important for selecting alternate attestation activities based on system configuration, error handling, or other exceptional requirements of the target system.

Coq supports a unique set of features that make it a convenient platform for our framework. First, it has robust inductive datatypes with corresponding inductive proof principles. This allows us to build protocols and reason about them inductively. Inductive datatypes in Coq also provide a natural way to encode mathematical relations, which in turn are a natural way to encode evaluation rules for a language. We exploit this feature when defining the operational semantics of protocol execution. Next, Coq has a rich functional programming environment where we can simulate local operations on protocol data. Finally, and perhaps most importantly, Coq has expressive verification capabilities with dependent types and proofs as first class structures. We can pass the protocols *themselves* as arguments to a proof, pass proofs *about* a protocol to other proofs, etc. Coq's interactive nature allows us to discover proof patterns over protocol properties, while its automation capabilities can capture these patterns and attempt to discharge them automatically. We are also encouraged by Coq's *code extraction* capabilities [Leroy, 2012, Ono et al., 2011, Carette et al., 2006]. However, we leave integration of these verified protocols into an emerging Haskell remote attestation framework [ArmoredSoftware, 2016] for future work.

If we view attestation protocols as programming language expressions, we can borrow from standard language semantics techniques to model their execution. We can think of a protocol as a program where the data values are cryptographic evidence packages, and communication events and branching notifications act as functions and control structures. We can think of a protocol as being *evaluated* to produce a result. In fact, we can prove that a well-typed protocol will always produce an evidence package of a particular form. We model evaluation using the popular small-step operational semantics approach.

One unique characteristic of a "protocol-as-a-language" is that since a protocol involves two communicating participants, it is made up of two distinct *halves*. In effect, the two participants collaborate to evaluate the protocol in tandem. Although evaluation requires both halves of the protocol, each half may be built independently and combined freely with many different counterparts. This compositional approach promotes the re-use of protocol *snippets* and creates the exciting opportunity for compositional verification. These snippets are readily paramaterizable, and can be used in an off the shelf, protocol-as-a-library fashion. Thus, in addition to the general properties of protocol execution that hold for *all* protocols, we can prove specialized properties of individual protocols or families of protocols. Coq is an ideal exploratory space to develop these custom proof strategies.

Chapter 2

Background

2.1 Remote Attestation

Remote Attestation [Haldar et al., 2004] is the process by which a target system attests to a particular claim about its operational state. An *attestation* agent on the target must collect and report evidence to a *remote* appraiser in support of the claim. The appraiser examines this evidence against its own standards before deciding whether or not it trusts the target system. An inherent conflict exists between the target's desire to protect itself and the appraiser's desire to learn about the target. Even when the participants agree on the information they will share, it is difficult to verify that the goals of attestation are met while protecting sensitive information.

Coker et al. [2011] refine the remote attestation process centering on the selection and execution of an *attestation protocol* on a remote target. The attestation protocol sequences execution of *attestation service providers (ASPs)* that: (i) measure running applications; (ii) gather stored evidence; (iii) interact with a *trusted platform module* (TPM) or virtualized TPM; and (iv) perform nested attestations.

Attestation protocols use cryptography to protect sensitive information and also as a means of meta-evidence to authenticate the attestation process. A trusted platform module (TPM) provides strong identification and other cryptographic services during attestation, and can act as a root of trust [Martin et al., 2008]. Virtualized platforms are increasingly appealing platforms for remote attestation because they provide domain separation and are widely available on stock hardware.

A canonical example of an attestation protocol is the Needham-Schroeder protocol [Needham and Schroeder, 1978]. Figure 2.1 provides an *informal* description of the protocol as a message sequence diagram. The representation is informal because it cannot be checked by a computer. Message sequence diagrams also suffer from their inability to specify intermediate local operations, such as decryption, during protocol execution. The goal of Needham-Schroeder is to exchange secret nonces with a trusted peer. `A` and `B` are participants in the protocol, and an arrow from A to B specifies a message sent from A to B. Later in this thesis, we describe the protocol in more detail, represent it *formally*, and prove an important property of authentication.

$$
\begin{aligned}
&A \rightarrow B : \{N_A\}_{B+} \\
&B \rightarrow A : \{N_A, N_B\}_{A+} \\
&A \rightarrow B : \{N_B\}_{B+}
\end{aligned}
$$

Figure 2.1. message sequence diagram for Needham-Schroeder

2.2 Cryptography

Cryptography is a powerful tool that enables secure communication in the prescence of malicious participants in a protocol. At the core of cryptography are *cryptographic keys* used to perform encryption and decryption over messages. Encryption and decryption are inverse operations: encryption hides the content of a message while decryption uncovers the content of a previously encrypted message. Since keys are implemented as large randomly generated numbers, it is reasonable to assume they cannot be guessed. A *nonce* is also a large, effectively unique number used to indicate freshness and prevent replay attacks. A *hash* is a one-way function that produces a unique representative value of its message input. A *signature* is the encrypted hash of a message that binds it to the owner of the signing key.

Asymmetric cryptography is a popular cryptographic scheme that enables both secrecy and authentication. It involves a pair of keys: a private key and a public key. The private key is often bound and protected under a single platform, and it performs decryption and provides strong authentication via cryptographic signatures. The public key is often publicly available, and performs encryption and signature checking. We say that the public key member of the pair is the *inverse* of the private key member, and vice versa. If a message is encrypted with a key k, the *only* key that can decrypt it is the inverse of k. *Symmetric cryptography* involves a single shared key that performs both encryption and decryption.

2.3 Session Types

Session Types are a type discipline that describe the structure of communication protocols. They were first introduced by Honda et al. [1998] as an extension of the pi-calculus. The main communication primitives that *inhabit* Session Types are `send`, `receive`, `choice`, and `offer`. `send` transmits a message and `receive` accepts a message. The Session Type for a protocol that first performs a `send` is `a :!: p` where `a` is the message type and `p` is the Session Type for the remaining communication primitives after the `send`. Similarly, the Session Type for `receive` is `a :?: p`. The `choice` primitive allows an entity to *choose* to proceed in accordance with one of two protocols. Its Session Type is `p1 :+: p2` where p1 and p2 are the Session Types of the two protocol options. Similarly, the Session Type for `offer` is `p1 & p2`, and represents that an entity is *prepared* to proceed in accordance with either `p1` or `p2`.

Assigning a Session Type to each communication primitive ensures that a sequence of such primitives conforms to its specification. If two participants in a protocol have compatable Session Types, they share a notion of the protocol structure; they know what types of messages to expect from one another, and when to expect them. In the Session Types literature, two participants that are compatable in this way are called *dual*. The Session Type of one participant is dual with the Session Type of another only if each `send` aligns with each `receive`, each `choice` aligns with each `offer`, and when a `send` aligns with a `receive` they agree on the message type. If we view protocols as programming language expressions, two participants that are dual represent a *well-typed* protocol.

We can think of Session Types as *indexing* communication primitives: as we build a protocol, we are also building its Session Type. In our Coq representation, dependent types capture this definitional style naturally: the Session Type of a protocol depends on its messages and sub-protocols. Because Session Types are resolved *statically*, we can verify that two protocol participants are well-typed prior to execution. A more recent implementation of Session Types in Haskell [Pucella and Tov, 2008] confirms the feasibility of inferring Session Types statically in a high-level language. This thesis explores Session Types in the context of attestation protocol verification where they guide the structure of proofs about protocol execution. They also monitor the structure of messages exchanged and guarantee that protocols produce evidence packages of the correct type.

Chapter 3

Protocol Representation

Before we discuss protocol execution, we must have some suitable representation of the protocols themselves. The representation must provide a way to build independent halves of a protocol, on behalf of a single executing entity, while also recognizing that entities share data during execution. A protocol expression should consist of dual communication primitives that account for the complex cryptographic structure of the messages exchanged. To accomplish this in our representation, we index protocol expressions with Session Types, where the `Send` and `Receive` Session Types are themselves indexed by cryptographic message types. We can easily encode each of these components as Coq inductive types, giving us all the benefits of the Coq verification environment. In this chapter, we will use the terms *protocol* and *protocol expression* to describe both a single participant involved in a protocol, and also to describe a pair of communicating participants. It will be clear from the context which we mean. When necessary, we will use the terms *protocol half* or *entity* to make it explicit that we mean a single participant.

3.1 Message Types

A motivating factor in the task of formally understanding remote attestation protocols is the complexity inherent in the cryptographic messages that they assemble and deliver. These *evidence packages* are often a dense collection of primitive data values and cryptographic keys bundled in arbitrary layers of encryption. The encryption may serve to hide private contents of the messages, or to strongly identify the source of the message. However, there are subtle ways to extract sensitive information from these messages, even when the protocols that transmit them are used as intended [Lowe, 1995, Paulson, 1998]

Our dependently typed model of perfect cryptography encoded as a Coq inductive type is based on a model by Lawrence Paulson in the Isabelle language [Paulson, 1998]. The term *perfect* in perfect cryptography means that we assume it is impossible to guess secrets. For instance, the only way a participant can attain a secret `key_val` of another participant is if it arrives as a message itself, or can be gleaned from an appropriate sequence of messages and decrypt operations. Our model aims to organize and monitor the structure and content of the messages exchanged by attestation protocols, all while maintaining the flexibility and utility of these complex bundles.

Let's start with the definition of `keyType` (Figure 3.1) that represents both symmetric keys and asymmetric key pairs. Each key holds its own `key_val` that we choose to be a natrual number. The definition of `inverse` (Figure 3.2) is not surprising: A symmetric key is its own inverse, and a public key is the inverse of the private key with the *same* `key_val` and vice versa. We can prove that this key representation exhibits properties from real cryptosystems. For instance, all keys must have an inverse, and the inverse must be unique (Figure 3.3). The proof of this statement follows from how we represent keyTypes using natural number key_vals.

```
Definition key_val : Type := nat.

Inductive keyType: Type :=
| symmetric : key_val → keyType
| private : key_val → keyType
| public : key_val → keyType.
```

Figure 3.1. keyType definition

```
Fixpoint inverse (k:keyType) : keyType :=
  match k with
  | symmetric k ⇒ symmetric k
  | public k ⇒ private k
  | private k ⇒ public k
  end.
```

Figure 3.2. inverse definition

```
Theorem inverse_bijective : ∀ k k',
      inverse k = inverse k' → k = k'
  ∧ ∀ k, ∃ k'', inverse k = k''.
```

Figure 3.3. A property of keys

The message type (Figure 3.4) represents information exchanged during protocol execution. Notice that message is indexed by a parameter of type type. type is its own inductive datatype, and defines our type language for messages. It should not be confused with the built-in Coq keyword Type (with an uppercase T) that is a universe holding other Coq datatypes. Each constructor of message is defined on its own line, where each line ends with a Coq expression of the form message <type>. This assigns the type to the message being built.

```
Inductive type : Type :=
| Basic : type
| Key : type
| Encrypt : type → type
| Hash : type
| Pair : type → type → type.
Inductive message : type → Type :=
| basic : nat → message Basic
| key : keyType → message Key
| encrypt (t:type) : message t → keyType → message (Encrypt t)
| hash : ∀ t, message t → message (Hash)
| pair : ∀ t1 t2, message t1 → message t2 → message (Pair t1 t2)
| bad : ∀ t1, message t1.
```

Figure 3.4. message type definition

This style of datatype definition is akin to GADTs in Haskell. In essence, we are defining the typing rules simultaneously with the syntax. In a more standard treatment[Pierce, 2002], terms of a language are defined independently of their types. Typing rules connect them by building a *type derivation*: a proof that a term has a certain type. If such a proof does not exist, the term is ill-typed. In our representation, building a term is the same thing as building its type derivation. Thus, if we can build a `message` term, it is well-typed *by construction*. For example, the only way to build a `message` of type `Encrypt t` is by providing a well-typed `message` of type `t` and a valid `keyType`. It is important to note that there is only *one* way to build each message type: through its sole constructor.

The constructors for primitive messages are `basic` and `key`. They are primitive in the sense that they do not hold other messages. `key` is simply a label for a `keyType` as described above. `basic` holds a natural number payload representing a primitive measurement in an attestation protocol. For simplicity, we assume all basic measurements are natural number values, but we could easily extend our model to account for other basic measurement types. Notice that the types of the primitive messages are `message Basic` and `message Key`, respectively.

The first compound message constructor is `pair`. `pair` is a standard product type that holds two messages of possibly different types. This is reflected in the final Coq expression of its definition, `Pair t1 t2`, where `t1` and `t2` are arbitrary message types. `pair` provides our first example of the expressive power of dependent types, since the type of the pair depends on its sub-messages. Notice that pairs can be arbitrarily nested, giving us the flexibility to return multiple measurements from a protocol, and also to represent more sophisticated measurement structures such as lists and trees, if desired.

The next compound message constructor is `encrypt`, representing a message encrypted with a key. It takes a message of an arbitrary `type t`, along with a `keyType`, and builds a message of type `Encrypt t`. This provides another example of dependent types in action: the overall type of the message, `message Encrypt t`, depends on the message we provide to the `encrypt` constructor. This provides a convenient way to keep track of the cryptographic structure of the messages as they are built; we cannot build an encrypted message without tagging its type with the type of its payload. The final compound message constructor is `hash`, representing the cryptographic hash of its payload message. We don't reference `hash` again in this thesis, but it highlights that the `message` type is highly extendable.

Finally, we include the message constructor `bad` to encapsulate all forms of errors while operating on message types. One example of such an error is when decryption fails on an encrypted message. This happens when we attempt to decrypt using a key that is not the inverse of the key used to encrypt the message. Notice that the `bad` constructor can inhabit *any* message type. As we will see later in the examples section, this allows us to maintain compatible Session Types in the presence of errors. One disadvantage of this error representation is that it does not provide a descriptive error diagnostic. For instance, a `bad` message can be decrypted multiple times and ultimately return a `bad` message. But it may be difficult to pinpoint exactly where the message "went wrong". We acknowledge that we may need a more sophisticated error handling mechinism–possibly an inductive error type that maintains error type information. We leave this for future work.

3.2 Session Types (protoType)

Now that we have a suitable representation for messages, let's begin to describe the representation for the attestation protocols that exchange these messages. In the next section we will present the protocol expressions themselves, but first we need to define their type language. Our type language is based on Session Types, and we borrow names and notation directly from the Session Types literature. Similar to the type language for message types, we encode the type language for protocols as a Coq inductive type named `protoType` (Figure 3.5) with a constructor for each element of the type language.

```
Inductive protoType : Type :=
| Send : type → protoType → protoType
| Receive : type → protoType → protoType
| Choice : protoType → protoType → protoType
| Offer : protoType → protoType → protoType
| Eps : type → protoType.
```

Figure 3.5. protoType definition

The `Send` and `Receive` type constructors have the same signature. They both require a message `type` and another `protoType`. The `type` represents the type of message being sent or received, and the `protoType` represents the structure of the remaining protocol actions after the send or receive. This recursive representation gives an implicit way to sequence individual protocol types. For instance, a valid `protoType` for a protocol expression that sends a `Basic` message, then receives a `Basic` message and terminates (returning a `Basic` message) is as follows: `Send Basic (Receive Basic (Eps Basic))`. The `Eps` constructor takes a `type` argument that simply signifies the type of message returned at the end of the protocol. We can represent these types more concisely with infix notations for `Send` and `Receive` as follows: `Basic :!: Basic :?: Eps Basic`.

The `Choice` and `Offer` constructors also have matching type signatures taking two `protoType` arguments. Together, they allow us to represent protocol branching. As we will see in the next section, the `ChoiceC` protocol expression includes a boolean selector that chooses either the left or the right protocol branch in order to proceed with execution. Notice that this boolean is not mentioned in the `protoType` for `Choice`, although it could be. We also have concise infix notations for `Choice` and `Offer`. For instance, a protocol expression that receives a `Basic` message, then chooses between two protocols has the following type: `Basic :?: (P1T :+: P2T)`, where `P1T` and `P2T` are the types of the two protocols. Similarly, a protocol that sends a `Basic` message, then is prepared to continue with two different protocols has the following type: `Basic :!: (P1T’ :&: P2T’)`.

3.3 Protocol Expressions (protoExp)

Now that we have the type language for protocols defined, let's define the protocol expressions that inhabit these Session Types. Once again, we use a Coq inductive type with a constructor for each communication primitive. Notice that `protoExp` (Figure 3.6) is indexed by a protoType argument that is its Session Type. Each constructor definition ends with a statement of the form `protoExp` <`protoType`> that assigns a Session Type to the expression.

```
Inductive protoExp : protoType → Type :=
| SendC {t:type} {p':protoType} : (message t) → (protoExp p')
    → protoExp (Send t p')
| ReceiveC {t:type} {p':protoType} : ((message t)→(protoExp p'))
    → protoExp (Receive t p')
| ChoiceC (b:bool) {r s:protoType} : (protoExp r) → (protoExp s)
    → (protoExp (Choice r s))
| OfferC {r s : protoType} : (protoExp r) → (protoExp s)
    → protoExp (Offer r s)
| ReturnC {t:type} : (message t) → protoExp (Eps t).
```

Figure 3.6. protoExp definition

Also notice that each constructor introduces arbitrary types using Coq's curly brace syntax for implicit types. Coq allows these types to be made implicit since it can always infer them from the values provided. This is an appealing property of our dependently-typed representation: as we build a protocol expression we are also building elements of its Session Type, and Coq can infer them.

The SendC constructor holds a message of arbitary `type t`, and a `protoExp` of an arbitrary `protoType p'`. `ReceiveC` may be the most interesting constructor, as it holds a *function* from an arbitrary message type to an arbitrary protocol expression. A function is a natural representation for the receive primitive: a function, like a listening protocol participant, waits for an input message, then proceeds with the remaining protocol with this new message in its scope. We can see the duality of SendC and ReceiveC since SendC provides the message that ReceiveC consumes.

The ChoiceC constructor holds a boolean selector, and also holds two arbitrary protocol expressions. The types of these two protocol expressions appear in the overall Choice Session Type. The OfferC constructor holds two arbitrary protocol expressions. The types of these two protocol expressions also appear in its Offer Session Type. As we will see later, during execution the OfferC expression depends on the boolean selector from ChoiceC to determine which of its branches to take. The ReturnC constructor simply holds an arbitrary return message, and signifies the last action of a protocol.

We can now build protocol expressions on behalf of a single executing entity. However, these expressions only account for one half of a protocol on their own. We need to define what it means for two halves of a protocol to be *compatible*. By compatible we mean that when the two halves combine, their corresponding communication primitives align in such a way that, together, they are capable of executing a protocol to completion. We say that two protocols are *dual* if they are compatible in this way. Dualness is a static property of protocol expressions, and as such we can define it as a recursive binary proposition over Session Types called DualT (Figure 3.7).

Notice that a Send is only compatible with a Receive when they agree on the message type and also when their remaining protocol types are themselves dual. This ensures that the receiving end knows the structure of the message sent, and that this "knowing of structure" holds for both participants for the rest

```
Fixpoint DualT (t t':protoType) : Prop :=
  match t with
  | Send p1T p1' ⇒
    match t' with
    | Receive p2T p2' ⇒ (p1T = p2T) ∧ (DualT p1' p2')
    | _ ⇒ False
    end
  | Receive p1T p1' ⇒
    match t' with
    | Send p2T p2' ⇒ (p1T = p2T) ∧ (DualT p1' p2')
    | _ ⇒ False
    end
  | Choice p1' p1'' ⇒
    match t' with
    | Offer p2' p2'' ⇒ (DualT p1' p2') ∧ (DualT p1'' p2'')
    | _ ⇒ False
    end
  | Offer p1' p1'' ⇒
    match t' with
    | Choice p2' p2'' ⇒ (DualT p1' p2') ∧ (DualT p1'' p2'')
    | _ ⇒ False
    end
  | Eps _ ⇒
    match t' with
    | Eps _ ⇒ True
    | _ ⇒ False
    end
  end.
```

Figure 3.7. DualT definition

of the protocol. `Choice` is only compatible with `Offer` when their left and right branches are duals of one another. This ensures that the protocols remain dual regardless of the branch selected. Finally, as a base case, `Eps` is only dual with another `Eps`. This ensures that when one side of the protocol finishes, so does the other. It is important to note that, by definition, any other pair of `protoType`s are *not* duals. We define the proposition `Dual` (Figure 3.8) over protocol expressions simply in terms of `DualT`.

```
Definition Dual {t t':protoType} (p1:protoExp t) (p2:protoExp t') : Prop
  := DualT t t'.
```

Figure 3.8. Dual definition

Chapter 4

Protocol Semantics

A protocol expression is a series of one-sided communication events that represent the actions of a *single participant* in a protocol. However, it doesn't make sense to evaluate a protocol expression on its own. In order to define the semantics of a protocol *in its entirety*, we must consider a *pair* of protocol expressions that collaborate to produce a result. For instance, a `ReceiveC` command must obtain its input message from an accompanying `SendC` command in order to proceed with execution. We present our evaluation semantics in the standard small-step operational style, and encode them naturally within a Coq Inductive datatype.

4.1 Single-step

We first define what it means for a protocol expression to take a single step in its execution. We achieve this by using a Coq Inductive datatype called `step` (Figure 4.1) that has a constructor for each evaluation rule. The type signature for `step` takes an argument of type `State`, three arbitrary `protoExp` arguments (preceded necessarily by their `protoType` index arguments), and another `State` argument. The first State argument represents the state *before* this step of execution. The last State argument represents the state *after* this step of execution. We will discuss the specifics of the State structure in more detail later, but a desirable property of our evaluation semantics is that we can leave State abstract. There is nothing in our semantic definition that depends on our implementation of State. The only way we modify State is through the updateState function, which we can also leave abstract. This allows us to continue to enhance the implementations of State and updateState without modifying the underlying evaluation semantics.

The first two protoExp arguments to step–indexed with protoTypes t and r, respectively–represent the two halves of the protocol we are evaluating. The third protoExp argument–indexed with protoType t'–represents the first protoExp argument after taking one step of evaluation. So in reality, we are only advancing the *first* protoExp argument a single step. The second protoExp simply *aids* in evaluation.

Let's look at the evaluation rules themselves. Each rule is a constructor with a name that starts with ST and introduces variables that it needs in its definition. For example, the rule ST_Send_Rec introduces a message m, a protoExp p1, a function f, and a State st. Notice that the rule also introduces the necessary index variables pt1, pt2, and mt in order to populate its other variables. Finally, the last line of each rule–starting with step–describes how two protocol expressions are reduced to one. For the ST_Send_Rec rule, a SendC paired with a ReceiveC reduces to the remaining protocol inside the SendC constructor, p1. For the ST_Rec_Send rule, the ReceiveC constructor is the leftmost expression, and it evaluates to the result of applying the function f to the message m supplied by

```
Inductive step : ∀ (stIn:State) (t r t':protoType),
    (protoExp t) → (protoExp r) → (protoExp t') → State → Prop :=
| ST_Send_Rec : ∀ pt1 pt2 mt
                  (m:message mt) (p1:protoExp pt1)
                  (f:(message mt) → protoExp pt2) (st:State),
    step st _ _ _ (SendC m p1) (ReceiveC f) p1 st
| ST_Rec_Send : ∀ pt1 pt2 mt
                  (m:message mt) (p1:protoExp pt1)
                  (f:(message mt) → protoExp pt2) (st:State),
    step st _ _ _ (ReceiveC f) (SendC m p1) (f m) (updateState m st)
| ST_Choice_true : ∀ pt1 pt1' pt2 pt2'
                     (r:protoExp pt1) (r0:protoExp pt1')
                     (s:protoExp pt2) (s0:protoExp pt2') (st:State),
    step st _ _ _ (ChoiceC true r s) (OfferC r0 s0) r st
| ST_Choice_false : ∀ pt1 pt1' pt2 pt2'
                     (r:protoExp pt1) (r0:protoExp pt1')
                     (s:protoExp pt2) (s0:protoExp pt2') (st:State),
    step st _ _ _ (ChoiceC false r s) (OfferC r0 s0) s st
| ST_Offer_true : ∀ pt1 pt1' pt2 pt2'
                     (r:protoExp pt1) (r0:protoExp pt1')
                     (s:protoExp pt2) (s0:protoExp pt2') (st:State),
    step st _ _ _ (OfferC r0 s0) (ChoiceC true r s) r0 st
| ST_Offer_false : ∀ pt1 pt1' pt2 pt2'
                     (r:protoExp pt1) (r0:protoExp pt1')
                     (s:protoExp pt2) (s0:protoExp pt2') (st:State),
    step st _ _ _ (OfferC r0 s0) (ChoiceC false r s) s0 st.
```

Figure 4.1. single-step evaluation relation

its SendC counterpart. This results in a protoExp indexed with Session Type pt2, as specified in the type of f. We can think of these evaluation rules as "peeling off" a SendC or ReceiveC.

The ST_Choice_true rule says that a ChoiceC expression where the boolean selector is true evaluates to its leftmost protocol expression, here r. The ST_Choice_false rule is similar, except the boolean selector is false and it evaluates to the rightmost expression, s. The boolean selector here is of the standard boolean type in

Coq, and is not an explicit part of our protocol language definition. Because of this, the selector may actually be a complex `boolean` expression. But we know that all Coq functions terminate [Coquand and Huet, 1988], so it will evaluate to `true` or `false` eventually. The evaluation rules for `OfferC` are similar to those for `ChoiceC`.

4.2 Multi-step

Next, we define the multi-step evaluation relation for protocol expressions as the reflexive, transitive closure of single-step evaluation. We encode the relation as a Coq Inductive datatype named `multi` (Figure 4.2).

```
Inductive multi : ∀ (stIn:State) (t r t':protoType),
    (protoExp t) → (protoExp r) → (protoExp t') → State → Prop :=
| multi_refl : ∀ (t r :protoType) (x:protoExp t) (y:protoExp r) (st:State),
    multi st _ _ _ x y x st
| multi_step : ∀ (t t' r r2 s:protoType),
    ∀ (x:protoExp t) (x':protoExp t')
      (y:protoExp r) (y':protoExp r2)
      (z1:protoExp s) st st' st'' st2 st2',
                      step st _ _ _ x x' y st' →
                      step st2 _ _ _ x' x y' st2' →
                      multi st' _ _ _ y y' z1 st'' →
                      multi st _ _ _ x x' z1 st''.
```

Figure 4.2. multi-step evaluation relation `multi`

The type signature for `multi` is identical to that of `step` because they are both relations from a pair of protocol expressions to another protocol expression. The difference is that `step` represents a single step of evaluation; `multi` represents zero or more steps. For single-step evaluation, we said that the rightmost protocol was there to *aid* in evaluation of the leftmost. However, for multi-step evaluation it is important that both sides of the protocol advance in lock-step fashion with one another. After the first step of multi-step evaluation, if the protocol is not already finished, both sides will again rely on eachother to continue with evaluation.

The multi_refl constructor defines reflexivity, acting as a base case in a proof of multi-step evaluation. It says that a protocol expression “evaluates” to itself in zero steps without modifying its State. The multi_step constructor is more interesting. It defines transitivity to chain single steps of evaluation. The definition of multi_step seems cluttered due to the variable introductions, but the last four lines are the most important. The last line shows our ultimate goal: a proof that the protocol expression x–paired with protocol expression x′ with an initial State of st–evaluates to a protocol expression z1 with a final state of st″. In order to prove this, we need three intermediate proofs: a proof that x–paired with x′ with State st–evaluates to y in State st′, a proof that x′–paired with x–evaluates to y′, and a proof that y–paired with y′ with State st′–evaluates to z1 in State st″. What is happening here is that first we evaluate both halves of the protocol–x and x′–one step, where each half aids the other. This gives us two new protocol halves y and y′. Then if we can evaluate y and y′ in zero or more steps to our goal expression z1, we're done.

4.3 Values and Normal Forms

When we view protocol expressions as programming language expressions, it is natural to define a value set, or a set of special expressions that we view as being the final result of evaluating a protocol. Our ReturnC expression is the obvious choice since it signifies the end of protocol execution. We can represent this easily in Coq as a proposition over protocol expressions called isValue (Figure 4.3).

```
Definition isValue {t:protoType} (p:protoExp t) : Prop :=
  match p with
  | ReturnC _ ⇒ True
  | _ ⇒ False
  end.
```

Figure 4.3. defining value set

Another important set of expressions in a language are *normal forms*. Normal forms are expressions that cannot be evaluated further. We defined the value set directly and somewhat arbitrarily. Normal forms, however, are a built-in property of the language, and they arise from a standard definition. `normalForm` (Figure 4.4) is a proposition over a *pair* of protocol expressions. It says that for protocol expressions `p1` and `p2`, and for all starting and ending `States`, there is *no* protocol expression `x` such that `p1` evaluates to `x` with the help of `p2`.

```
Definition normal_form {p1t p2t:protoType}
                (p1:protoExp p1t)(p2:protoExp p2t) : Prop :=
  ∀ st st', ¬ ∃ t' (x:protoExp t'), step st _ _ _ p1 p2 x st'.
```

Figure 4.4. normal form definition

Some languages have an appealing property that their value set and normal form set are identical. When this is not the case, it is possible for some terms to be *stuck*: a normal form but not a value. Stuck terms are usually undesirable because our operational semantics does not know how to handle them. In a sense, the program has reached a "meaningless state" Pierce [2002]. In our protocol language values and normal forms are one and the same. We prove this and other desirable properties of our protocol language in the next chapter.

Chapter 5

Semantics Proofs

The multi-step evaluation relation for our protocol representation consists of rules by which a pair of protocol expressions may proceed in evaluation. Although these rules seem reasonable, they don't come with any formal guarantees about the nature of evaluation. In particular, it isn't immediately obvious that a pair of protocol expressions remains well-typed throughout execution. Remember, the types of protocol expressions are Session Types, and a pair of well-typed protocol expressions must have `Dual` Session Types. If we again view protocols as programming language expressions, we can prove several standard properties from the language semantics literature that are also pertinent for protocols. Type Safety is a critical goal, but protocol termination is also important: protocols should always run to completion and produce an evidence package of the correct type. Just as types are the backbone of such proofs about programming languages [Pierce, 2002], Session Types are the backbone of proofs about protocols.

A convenient way to build a proof in Coq is by using interactive proof commands known as *tactics*. Input arguments to the proof become the hypotheses that we manipulate with tactics in order to build a proof object of the result type. We limit our discussion of proof tactics to what is necessary to convey the structure of our proofs, but there are resources that provide a more thorough coverage of their full power and utility [Pierce et al., 2015, Chlipala, 2013, Coq development team, 2016]. The Coq interactive environment consists of a source code buffer and a goal buffer. The source code buffer contains Coq definitions, theorems, and proof scripts. The user proceeds interactively through the source code in a linear fashion. The goal buffer shows the state of the current proof in focus: a list of hypotheses above a solid line and a goal below. In the sections that follow we discuss the statement of each Theorem in detail, then outline the structure and interesting elements of each proof.

5.1 Progress

Towards the goal of Type Safety, we will first prove the property of *progress*: A well-typed term is either a value or it can take a step according to the evaluation rules.

```
Theorem progress {t t':protoType} :
  ∀ (p1:protoExp t) (p2:protoExp t') st,
    (Dual p1 p2) →
    isValue p1 ∨ (∃ t''(p3:protoExp t'') st', step st _ _ _ p1 p2 p3 st').
```

Figure 5.1. statement of `progress` theorem

The type signature for the `progress` `Theorem` (Figure 5.1) takes two protocol expressions `p1` and `p2` of arbitrary Session Types, an arbitrary initial `State` `st`, a proof argument of type `(Dual p1 p2)`, and returns a proof of the `progress` property. In particular, it returns a proof of the disjunction that either `p1` is a value, or there exists a protocol expression `p3` and final `State` `st'` such that p1 in State st can take a step to p3 in State st' with the help of p2. It is no coincidence that the syntax for Theorem types is identical to the syntax for function types in Coq: building a proof to inhabit a Theorem type is the same as building a function to inhabit a function type. This is one of many manifestations of the Curry-Howard Correspondence [Wadler, 2015] in Coq. The Curry-Howard Correspondence draws convincing parallels between proofs and programs, as well as between theorems and types.

Here we present a completed proof script of the progress Theorem as a series of proof tactics enclosed by the Coq keywords Proof and Qed (Figure 5.2). Comments in proof scripts are enclosed by the symbols (* and *).

```
Proof.
  intros p1 p2 st dualProof. destruct p1; destruct p2; inversion dualProof.
  (*  Case:  p1 = SendC, p2 = ReceiveC *)
  right. subst.
    ∃ p1'. eexists. constructor.
  (*  Case:  p1 = ReceiveC, p2 = SendC *)
  right. subst.
    ∃ (f m). eexists. constructor.
  (*  Case:  p1 = ChoiceC, p2 = OfferC *)
  right. destruct b.
    ∃ p1_1. eexists. constructor.
    ∃ p1_2. eexists. constructor.
  (*  Case:  p1 = OfferC, p2 = ChoiceC *)
  right. destruct b.
    ∃ p1_1. eexists. constructor.
    ∃ p1_2. eexists. constructor.
  (*  Case:  p1 = ReturnC, p2 = ReturnC *)
  left. simpl. trivial.
Qed.
```

Figure 5.2. proof of progress theorem

The first line of tactics (after the Proof keyword) outlines the structure of this proof. The intros tactic brings the universally quantified variables and the (Dual p1 p2) proof into the context so that they are available as hypotheses. For clarity we provide variable names p1, p2, and dualProof to the intros tactic, although Coq would generate suitable names for us automatically. Next, the destruct tactic performs proof-by-cases. When we apply destruct to the protocol expression p1, it generates a new sub-goal for each constructor of protoExp, replacing p1 with a representative value of the constructor in each case. Since protoExp has five constructors, destruct p1 generates five new sub-goals. destruct works on any Inductive datatype. The ; operator is a binary infix operator that performs the first tactic, then performs the second tactic on *each sub-goal* generated by the first. Thus, the compound command destruct p1; destruct p2 generates 25 subgoals representing all the possible forms that a pair of protocol expressions may take. ; is one of many *Tacticals* in Coq: a tactic that takes other tactics as arguments.

Obviously, some of these pairs are not well-typed. For example, a pair of SendC expressions are not Dual. In fact, there are only five well-typed pairs: (SendC, ReceiveC), (RecieveC, SendC), (OfferC, ChoiceC), (ChoiceC, OfferC), and (ReturnC, ReturnC). What we would like is a way to eliminate all sub-goals where the pair of protocol expressions are ill-typed. The inversion tactic does exactly this. When p1 and p2 are *not* Dual, inversion recognizes the contradiction in the dualProof assumption and discharges the sub-goal automatically. In the cases where p1 and p2 *are* Dual, rather than discharging the sub-goal immediately, inversion adds new assumptions that help prove progress in each case. These new assumptions arise from what it means to be Dual. For instance, if p1 is (SendC m p1') and p2 is (ReceiveC f) where m has type (message t) and f has function type (message t0 → protoExp p'0), inversion dualProof generates an assumption that `t = t0`. Remember, the `destruct` tactic populated `p1` and `p2` with representative values, including the message types `t` and `t0` which may or may not be equal. `t = t0` is an important additional assumption in proving `progress` because in order to take a step of evaluation under the `step` relation, `SendC` and `ReceiveC` must agree on their message type.

In the first well-typed sub-goal, `p1` is `SendC` and `p2` is `ReceiveC`. Since our goal is a disjunction, we must prove only *one* of its members. In this case, the `right` tactic tells Coq that we intend to prove the rightmost member–i.e. that the protocol can take a step. Next, the `subst` tactic uses the `t = t0` assumption–generated earlier by `inversion`–to perform substitution and unify the message types. Finally, we must provide a witness `protoExp`, along with its `protoType` index and starting and ending `State`s, such that our pair of protocol expressions take a step to it. In this case, our leftmost `protoExp` is a `SendC` that holds another `protoExp`, which Coq named `p1'`. This `p1'` is the witness we need–it represents the remainder of the protocol after the send. We provide this witness with the ∃ tactic, then let the `eexists` tactic infer the witness for the ending `State`. Finally, the `constructor` tactic looks for a constructor of `step` that matches our goal. Since the `ST_Send_Rec` constructor is such a proof, the sub-goal is discharged. The next sub-goal where `p1` is `ReceiveC` and `p2` is `SendC` is proved in an identical way, except that the `protoExp` witness is `(f m)` where `f` is the function held by `ReceiveC`. `ReceiveC` can take a step by applying `f` to the message `m` provided by `SendC`. `(f m)` is the remainder of the protocol after the receive.

The cases for `ChoiceC` and `OfferC` are similar, except we must `destruct` the boolean selector `b` from `ChoiceC`. In the sub-goal where `b` is `true`, we provide the leftmost `protoExp` branch–named `p1_1` by Coq–as the next-step witness. When b is false, we choose the rightmost protoExp branch, p1_2. The final well-typed sub-goal occurs when both protocol expressions are ReturnC. In this case, we must prove p1 is a value. The left tactic chooses the isValue portion of the disjunction in the goal. Proving ReturnC is a value follows trivially from simplification using the simpl tactic.

5.2 Preservation

Next we verify the other half of Type Safety, namely *preservation*: *If* a well-typed term takes a step, the resulting term is well-typed.

```
Theorem preservation {t t' p3t p4t : protoType} :
    ∀ (p1:protoExp t) (p2:protoExp t'),
    (Dual p1 p2) →
    ∀ (p3:protoExp p3t) (p4:protoExp p4t) st st' st2 st2',
      (step st _ _ _ p1 p2 p3 st' ∧
       step st2 _ _ _ p2 p1 p4 st2')
      → (Dual p3 p4).
```

Figure 5.3. statement of preservation theorem

The type signature of preservation (Figure 5.3) takes a proof that protocol expressions p1 and p2 are well-typed, a proof that p1 and p2 can take a step of evaluation to protocol expressions p3 and p4, and returns a proof that p3 and p4 are well-typed. The ∀ quantifiers are critical: they express that we must consider *every* possible way a well-typed protocol can take a step. The overall structure of the preservation proof (Figure 5.4) is the same as for the proof of progress. In fact, the first line of proof tactics is identical.

The second line of proof tactics performs most of the legwork. The Coq try Tactical takes a series of tactics as an argument, and tries them on each sub-goal.

```
Proof.
  intros p1 p2 dualProof. destruct p1; destruct p2; inversion dualProof;
  try (intros; destruct H1; dep_destruct H1; dep_destruct H2; assumption).
  intros. destruct H. inversion H.
Qed.
```

Figure 5.4. proof of preservation theorem

try is useful for proof automation in conjunction with the ; Tactical because try does not fail or cause a proof to halt. It simply proves as much of each sub-goal as possible. In this case, the try phrase discharges four out of the five well-typed sub-goals. Using the dep_destruct [Chlipala, 2013] tactic on the step proof assumptions resolves p3 and p4 in our goal to the results of stepping p1 and p2, respectively. For example, if we know that p1 is (SendC m p1') and p2 is (ReceiveC f), then performing dep_destruct on the step st ___ p1 p2 p3 st' assumption resolves p3 to p1'–the remaining protocol after the send. We won't go into the details of the dep_destruct tactic here, but it relies on a version of destruct specialized for dependent types. Once we know the precise forms of p3 and p4, we find that a proof of (Dual p3 p4) already exists in our assumptions due to the recursive component of the earlier inversion on dualProof. For the final well-typed sub-goal where both protocol expressions are ReturnC, we can discharge it with inversion since Coq recognizes the contradiction in assuming that ReturnC can take a step: there is no such rule in the step relation.

5.3 Normalization

An attestation protocol that runs forever is of little use. It should eventually terminate and return a representative evidence package so that an external entity may check this evidence. Although Type Safety guarantees that a well-typed protocol remains well-typed throughout its evaluation, it does *not* guarantee termination. Thus, we must prove a separate property called *normalization*: All well-typed pairs of protocol expressions *must* evaluate to some `normal_form`. Remember, a `normal_form` is a pair of protocol expressions that cannot take a step of evaluation. The type signature of `normalization` (Figure 5.5) takes protocol expressions `p1` and `p2` along with a proof that they are well-typed, and returns a proof that they evaluate in zero or more steps to some `normal_form` pair `p3` and `p4`. Evaluation here is the multi-step evaluation relation `multi`. The ∀ and ∃ quantifiers ensure that *all* well-typed protocols have *some* normal form under the `multi` relation.

```
Theorem normalization {p1t p2t :protoType} :
    ∀ (p1:protoExp p1t) (p2:protoExp p2t),
      (Dual p1 p2) →
    ∃ p3t p4t (p3:protoExp p3t) (p4:protoExp p4t) st st' st2 st2',
      (multi st _ _ _ p1 p2 p3 st') ∧ (multi st2 _ _ _ p2 p1 p4 st2')
      ∧ normal_form p3 p4.
```

Figure 5.5. statement of `normalization` theorem

The proof of `normalization` (Figure 5.6) is similar in structure to the Type Safety proofs, except that we must use `induction` over the first protocol expression instead of `destruct` due to the recursive nature of multi-step evaluation. In addition to performing proof by cases over the constructors of an `Inductive` type, the `induction` tactic also generates an inductive hypothesis for every case where the constructor has a recursive sub-term. Recall that every constructor of `protoExp` except for `ReturnC` holds a recursive `protoExp` sub-term that represents the *rest* of its protocol actions. Thus, `induction p1` generates an induction

```
Proof.
  intros p1 p2 dualProof.
  induction p1; destruct p2;
  try (inversion dualProof).
  ...
  (*  step Case *)
  constructor.
  (*  inductive Case *)
  apply IHp1.
  ...
Qed.
```

Figure 5.6. fragment of proof of `normalization` theorem

hypothesis ensuring that `normalization` holds for all `protoExp` sub-terms of `p1`. For example, when `p1` is `(SendC m p1')` and `p2` is `(ReceiveC f)`, the inductive hypothesis `IHp1` has the type shown in Figure 5.7.

```
IHp1 : Dual p1' (f m) →
       ∃
         (p3t p4t : protoType)
         (p3 : protoExp p3t) (p4 : protoExp p4t)
         (st st' st2 st2' : State),
       (multi st _ _ _ p1' (f m) p3 st') ∧ (multi st2 _ _ _ (f m) p1' p4 st2')
       ∧ normal_form p3 p4
```

Figure 5.7. induction hypothesis for `normalization`

Recall that a proof of `multi` requires two pieces of evidence: 1) a proof of the first step of evaluation and 2) a proof of multi-step evaluation for the *rest* of the protocol. For each `multi` proof in the well-typed sub-goals of `normalization`, we can prove 1) with an appropriate constructor from the `step` relation. For 2), the induction hypothesis is exactly the proof we need. The induction hypothesis also discharges the `normal_form p3 p4` proof since the result of the *rest* of a protocol is the same as the result of the entire protocol.

5.4 Normal Form iff Value

Finally, we prove that each normal form is a value and vice-versa. This is a desirable property of our protocol language because it means that no terms can be *stuck*; a normal form but not a value. It also means that values cannot be evaluated further. First we state and prove that a value is a normal form.

```
Lemma value_is_nf {t t':protoType} (p1:protoExp t) (p2:protoExp t') :
  (isValue p1) ∧ (isValue p2) → normal_form p1 p2.
Proof.
  intros.
  destruct p1; destruct p2;
  (cbv; intros; solve by inversion 3).
Qed.
```

Figure 5.8. proof that a value is a normal_form

The type signature of `value_is_nf` (Figure 5.8) is fairly straightforward: if `p1` and `p2` are both values, then as a pair they are a `normal_form`. The proof (Figure 5.8) is fairly compact with the help of Coq's automation. The `cbv` tactic performs call-by-value evaluation to expand the definition of `normal_form`, then `solve by inversion` [Pierce et al., 2015] attempts to discharge the goal by performing `inversion` and substitution. Since `inversion` may generate new hypotheses, the number argument provided to `solve by inversion` tells Coq how many layers of nested `inversion` calls to attempt. In this case, the most we need for any sub-goal is 3.

The proof of the `nf_is_value` `Lemma` does not use any exotic features or proof structures, so we omit it here. The two `Lemma`s together prove the equivalence of normal forms and values in the `nf_same_as_value` Corollary (Figure 5.9). The `apply` tactic allows us to use previously proven results in a proof script. The Coq labels of Lemma and Corollary are simply meant as documentation of intent; they behave just like a Theorem.

```
Corollary nf_same_as_value {t t':protoType} :
  ∀ (p1:protoExp t) (p2:protoExp t'),
    (Dual p1 p2) →
    normal_form p1 p2 ↔ (isValue p1) ∧ (isValue p2).
Proof.
  intros. split.
  intros. apply nf_is_value in H0. assumption. assumption.
  intros. apply value_is_nf in H0. assumption.
Qed.
```

Figure 5.9. proof of value and normal form equivalence

Chapter 6

Example Protocols and Analysis

The protocol representation and execution semantics provide a general framework that imposes structure and formalizes *how* protocols evaluate to a result. Within this framework, a proof that a protocol is well-typed guarantees a collection of essential general properties about its execution. However, to prove specialized properties of specific protocols–or families of protocols–we must exploit properties of the messages exchanged. In our current implementation of messages, the most interesting properties involve encryption and decryption. In what follows we define example protocols in our representation, encode specialized properties as Coq Theorems, then present their proofs as an interactive proof script. Although the execution semantics remain constant, the proof strategies and analysis techniques for specialized properties vary significantly from protocol to protocol. This makes the interactive environment of Coq an appealing exploratory space.

For convenience and readability while building concrete protocols, we define the `send` and `receive` Coq notations in Figure 6.1 as shorthand for the `SendC` and `ReceiveC` constructors of `protoExp`. The `;` symbol in each notation separates the individual protocol action from the *rest* of the protocol; it acts as a sequence operator. Also note that in the receive notation, x is a variable bound in the scope of the *rest* of the protocol p. fun is Coq's syntax for annonymous functions.

```
Notation “ ‘send’ n ; p ” := (SendC n p)
                               (right associativity, at level 60).
Notation “ x ← ‘receive’ ; p ” := (ReceiveC (fun x ⇒ p))
                                    (right associativity, at level 60).
```

Figure 6.1. protoExp Coq notations

6.1 A Simple Example

As a first example, consider the following protocol: Entity A sends a Basic message, Entity B receives the message, increments it, then sends it back to A. Finally, A receives the incremented message and returns it. Both halves of the protocol in our Coq representation are in Figure 6.2. incPayload is a simple function that takes a Basic message and returns another Basic message with an incremented natural number payload. Also notice that proto1A accepts an arbitrary Basic message m that it sends. The ability to parameterize protocols adds great flexibility to our representation, promoting a *protocol-as-a-library* style.

```
Definition proto1A (m:(message Basic)) :=
  send m;
  x ← receive;
  ReturnC (t:=Basic) x.
Definition proto1B :=
  x ← receive;
  send (incPayload x);
  ReturnC x.
```

Figure 6.2. increment protocol definition

Since `incPayload` only accepts `Basic` messages, it enables Coq to infer the Session Type of `proto1B` by restricting the message type of `x` to `Basic`. We solve this same problem in `proto1A` by annotating `ReturnC` with `(t:=Basic)` to prescribe the message type of `x`. We print the inferred Session Types of `proto1A` and `proto1B` in Figure 6.3 using `Check`, the native Coq typechecking command. As expected, `proto1A` and `proto1B` are `Dual`, as stated in Figure 6.4. The ∀ quantifier before `x` ensures that `proto1A` and `proto1B` are `Dual` regardless of the `message` parameter passed to `proto1A`. The omitted proof of `dual1AB` follows from simplification.

```
proto1A
: message Basic →
   Basic :!: (Basic :?: Eps Basic)

proto1B
:  Basic :?: (Basic :!: Eps Basic)
```

Figure 6.3. result of `Check proto1A`, `Check proto1B`

```
Theorem dual1AB : ∀ (x:message Basic), Dual (proto1A x) proto1B.
```

Figure 6.4. statement that `proto1A` and `proto1B` are `Dual`

Because the Session Types of the two halves are `Dual`, this protocol automatically exhibits the desirable general properties proved in the last chapter. However, we can also prove properties specific to this protocol. In particular: Entity A should return the incremented result of the message it sends. Because Entity A receives `x` and immediately returns it, this property could also be read as: Entity A *expects* Entity B to increment the message. This property is encoded as a Coq Theorem called `incPropertyAB` in Figure 6.5. The ∀ quantifier ensures incPropertyAB holds for all input messages x to proto1A and all starting and ending States. x' is constrained as the result of evaluating the protocol, and is wrapped in a ReturnC since multi expects a protoExp as the result of evaluation. The proof of this property follows from using dep_destruct to piece apart the multi assumption. The repetition in this proof is ripe for automation, which we leave for future work.

```
Theorem incPropertyAB : ∀ x x' st st',
    multi st _ _ _ (proto1A x) proto1B (ReturnC x') st' →
    x' = incPayload x.
Proof.
  intros x x' st st' multiProof.
  dep_destruct multiProof as (stepL, stepR, rest).
  dep_destruct stepL. dep_destruct stepR.
  dep_destruct rest as (stepL', stepR', rest').
  dep_destruct stepL'. dep_destruct stepR'.
  dep_destruct rest'.
    reflexivity.
    inversion stepL'.
Qed.
```

Figure 6.5. statement and proof of incPropertyAB

6.2 Needham-Schroeder Definition

For a more involved example that incorporates cryptography, let's move to the Needham-Schroeder protocol [Needham and Schroeder, 1978]. The goal of Needham-Schroeder is that by the end of the protocol, Entity A will know Entity B's secret nonce and Entity B will know A's. It is also important that these nonces are protected by encryption such that no other entity can learn them. The protocol starts when Entity A sends its nonce to Entity B encrypted with B's public key.

B receives this message, attempts to decrypt it with its private key, then sends a pair message encrypted with A's public key back to A. The first element of this pair is the result of decrypting the message from A. If decryption is successful, this is A's nonce and otherwise it is the `bad` message. The second element of the pair is B's own nonce. Upon receiving this message, A tries to decrypt it with its private key. Then A extracts the second element from the pair–B's nonce if decryption was successful, the `bad` message otherwise–and re-encrypts it with B's public key before sending it back to B. B decrypts this message and returns it in a pair along with A's nonce that it learned earlier. A also returns a pair including the nonce it learned from B and its own nonce that it received encrypted from B. In summary, both entities return a pair that consists of 1) the nonce they learned from the other entity and 2) the other entity's notition of their own nonce.

The Coq representation of Needham-Schroeder (Figure 6.6) is clearer than the english description above. Notice that both halves of the protocol are parameterized by keys they use throughout execution. Each takes a private key named `myPri` for decryption and a public key named `theirPub` for encryption. The `decryptM` operation (Figure 6.7) takes an encrypted message and the key to use for decryption as arguments. Upon successful decryption it returns the message held by the encrypted message, otherwise it returns the `bad` message. Either way the protocol can continue in its execution, and Session Types remain compatable: the `bad` message can inhabit any message type. The expressive type of the input message `m` to `decryptM` ensures *statically* that decryption acts exclusively on encrypted messages.

`decryptM` actually invokes another function, `decrypt` (Figure 6.7), that has an interesting type signature. `decrypt` returns a Coq `sumor` type, a value of which

```
Definition Needham_A (myPri theirPub:keyType) :=
  send (encrypt _ aNonce theirPub);
  x ← receive;
  let y := decryptM x myPri in
  let y' := (pairFst (t1:=Basic) (t2:=Basic) y) in
  let y'' := (pairSnd (t1:=Basic) (t2:=Basic) y) in
  send (encrypt _ y'' theirPub);
  ReturnC (pair _ _ y' y'').

Definition Needham_B (myPri theirPub:keyType) :=
  x ← receive;
  let y : (message Basic) := decryptM x myPri in
  send (encrypt _ (pair _ _ y bNonce) theirPub);
  z ← receive;
  let z' := decryptM z myPri in
  ReturnC (pair _ _ y z').
```

Figure 6.6. definition of Needham-Schroeder

```
Definition decrypt{t:type} (m:message (Encrypt t)) (k:keyType) :
  (message t × is_decryptable m k) + {(is_not_decryptable m k)} :=
    ...

Definition decryptM{t:type} (m:message (Encrypt t)) (k:keyType) :
  message t :=
    match decrypt m k with
    | inleft (m',_) ⇒ m'
    | inright _ ⇒ bad t
    end.
```

Figure 6.7. definition of `decryptM`

can inhabit a normal value or a proof value. By convention, the type on the left of the `+` symbol describes success and the proof on the right describes an error. In this case, upon success `decrypt` returns a Coq pair with the decrypted message and a proof that `m` is decryptable with the key `k`. Upon failure, it returns a proof that `m` is not decryptable with `k`. This is another example of the expressive power of dependent types: input *values* m and k are used later to populate the type signature.

Notice that decryptM relies on the result of decrypt m k. Upon success, it extracts the decrypted message out of the pair and returns it. Upon failure, it ignores the is_not_decryptable proof and returns the bad message. The is_decryptable and is_not_decryptable proofs (Figure 6.8) are not pertinent during protocol execution, but they give us useful information about the keys involved in encryption and decryption: namely whether they are inverses or not. Although decryptM hides these proofs during protocol execution, they still appear as important assumptions in the interactive proof environment when proving properties about protocols that perform decryption.

```
Definition is_decryptable{t:type}(m:message t)(k:keyType):Prop :=
  match m with
  | encrypt _ m' k' ⇒ k = inverse k'
  | _ ⇒ False
  end.
Definition is_not_decryptable{t:type}(m:message t)(k:keyType):Prop :=
  match m with
  | encrypt _ m' k' ⇒ k ≠ inverse k'
  | _ ⇒ True
  end.
```

Figure 6.8. definition of is_decryptable and is_not_decryptable

pairFst and pairSnd (Figure 6.9) are standard projection functions over message pairs, except that they propogate the *badness* of their input message. aNonce and bNonce (Figure 6.10) are nonces encoded as Basic messages that hold a secret value. In Needham_A (Figure 6.6) we provide a type annotation for the y variable so that Coq may infer the complete Session Type. Finally, dualNeedham (Figure 6.11) states that Needham_A and Needham_B are Dual regardless of the keys used for encryption/decryption, and the omitted proof follows from simplification.

```
Definition pairFst{t1 t2: type} (m:message (Pair t1 t2)) : message t1 :=
  match m in message t' return message (getP1Type t') with
  | pair _ _ m1 _ ⇒ m1
  | bad _ ⇒ bad _
  | _ ⇒ bad _
  end.
Definition pairSnd{t1 t2: type} (m:message (Pair t1 t2)) : message t2 :=
  match m in message t' return message (getP2Type t') with
  | pair _ _ _ m2 ⇒ m2
  | bad _ ⇒ bad _
  | _ ⇒ bad _
  end.
```

Figure 6.9. definition of pair projection functions

```
Definition aNonceSecret := 11.
Definition bNonceSecret := 22.
Definition aNonce := (basic aNonceSecret).
Definition bNonce := (basic bNonceSecret).
```

Figure 6.10. nonce definitions

```
Theorem dualNeedham : ∀ ka ka' kb kb',
    Dual (Needham_A ka kb') (Needham_B kb ka').
```

Figure 6.11. statement that Needham_A and Needham_B are Dual

6.3 An Authentication Property of Needham-Schroeder

Now that we have described and defined the components of Needham-Schroeder, let's prove an important property of authentication. In particular: Entity A can only learn B's nonce if A can strongly identify itself with its private key. We encode this property in Coq as the `needham_A_auth Theorem` (Figure 6.13). We assume that both entities know the other's public key prior to execution, and we fix B's private key as `bPri` (Figure 6.12). A's private key `k` remains abstract in `needham_A_auth` when it is passed as the `myPri` parameter to `Needham_A`.

```
Definition bPri := (private 2).

Definition aPub := (public 1).
Definition bPub := (public 2).
```

Figure 6.12. assumed keys for Entities A and B

```
Theorem needham_A_auth : ∀ (k:keyType) (x:message Basic) st st',
    multi
      st
      _ _ _
      (Needham_A k bPub)
      (Needham_B bPri aPub)
      (ReturnC (pair _ _ x bNonce))
      st'
    → (k = inverse aPub).
```

Figure 6.13. statement of `needham_A_auth Theorem`

The type signature for `needham_A_auth` takes a proof of multi-step evaluation where `Needham_A` learns `bNonce`, and returns a proof that `k` must be the inverse of `aPub`. Since B knows `aPub`, and the inverse of `aPub` is unique (Figure 3.3), another way to view this property is that B will not disclose its nonce to A when k is anything other than the unique private key counterpart of aPub. x is the first element of the pair in the result of evaluating Needham_A under multi. Thus, it represents A's nonce as relayed back from B. In this Theorem, we don't place any further restrictions on the value of x.

```
Proof.
  intros k x st st' multiProof.
  dep_destruct multiProof as (stepL, stepR, rest).
  dep_destruct stepL. dep_destruct stepR.
  ...
  unfold decryptM in x0.
  destruct (decrypt
           (encrypt _ (pair _ _ aNonce (basic bNonceSecret)) aPub)
           k) as [p | _].
    (* Case: decrypt succeeds *)
    destruct p as (m , is_dec_proof). dep_destruct is_dec_proof. reflexivity.
    (* Case: decrypt fails *)
    inversion x0.
  ...
Qed.
```

Figure 6.14. proof of needham_A_auth Theorem

The proof of needham_A_auth (Figure 6.14) starts by introducing variables with intros, then performs dep_destruct on the multi assumption and the assumptions it generates. The interesting part of the proof is when we perform destruct on Entity A's decrypt operation. The result of decrypt is a sumor type, and sumor is nothing more than an Inductive type with two constructors. Thus, destruct over decrypt generates two new subgoals: one for when decryption is successful and one for when it fails. The Coq as clause names the assumptions generated in each sub-goal.

We assign the name p to the Coq pair held by the left portion of the sumor and ignore the right portion by using an underscore. In the case where decryption succeeds, we extract the is_decryptable proof from p by using destruct and assign it the name is_dec_proof. destruct p does not generate any new sub-goals because Coq pairs are an Inductive datatype with only one constructor. We name the decrypted message m for clarity, although we could have ignored it with an underscore. Next, performing dep_destruct on the is_decryptable proof resolves the value of k in our goal to be inverse aPub as desired, and we discharge the sub-goal with reflexivity. In the case where decryption fails, we derive a contradictory assumption as follows: bad Basic = basic bNonceSecret, and discharge it with inversion: basic and bad are two distinct constructors of an Inductive datatype and cannot be equivalent. The bad Basic half of the equality arose from the definition of failed decryption, while the basic bNonceSecret half arose from our assumption that Needham_A successfully learns bNonce.

Chapter 7

Conclusion and Future Work

The contributions of this thesis are in the context of two-participant attestation protocols. Our protocol representation captures the structure of communication events and also the structure of the messages they exchange. Session Types are the critical organizational element in this representation. Not only do they ensure compatability amongst protocol participants, but they guide proofs of general properties of protocol execution. Coq is a natural environment for this representation because of its support for dependnet types. Dependent types ensure that building a protocol expression also builds its Session Type. Likewise, building a cryptographic evidence package with the `message` type ensures that the type of the message reflects how it was constructed.

In addition to the protocol representation, this thesis provides a semantic framework for attestation protocol execution. Our small-step operational semantics gives precise evaluation rules for protocols built in the representation. Guided by Session Types, we proved desirable properties of protocol execution such as Type Safety and termination. We also showed the feasibility of proving specialized properties of individual protocols. These proofs rely on properties of the messages they exchange, and proof strategies are often different from protocol to protocol. However, the Coq interactive proof environment is ideal for this style of exploratory proof. There are many avenues to explore for verification of specialized properties of attestation protocols. But the semantic framework in this thesis and the expressive capabilities of Coq provide a solid foundation for future work.

Real-world remote attestation protocols often involve multiple participants. For example, an attestation agent may need to interact with an external Certificate Authority to certify the keys it uses, a TPM for authentication and cryptographic services, or it may even invoke other appraisers to perform nested attestations. Our semantic framework for protocol execution is a solid foundation for understanding the behavior of two-participant sessions. However, to support multiparty attestation protocols there must be a way to chain these two-participant sessions. In future work, we envision a representation that sequences the execution of protocol sessions, where the resulting evidence package and final `State` from one session are made available to the next. The `State` structure could hold all messages that a participant learns directly or discovers through decryption during a session.

In the hypothetical notation presented in Figure 7.1, `doProto` performs multistep evaluation of an entire two-participant protocol, then binds its resulting evidence package and ending `State` to variables. In this example, `protoA1` and `protoA2` are protocol expressions that act on behalf of single entity A. First, `protoA1` has a session with `protoB` to produce evidence package `x` and final `State` `st1`. Continuing, `protoA2` consumes `st1` and `x`, then has a *seperate* session with `protoC` resulting in evidence package `y` and final `State` `st2`. The parameterizabil-

```
Definition multi_party_protocol{t t':protoType}
                 (protoB: protoExp t)
                 (protoC: protoExp t')
                 (d1: ∀ st, Dual (protoA1 st) protoB)
                 (d2: ∀ m st, Dual (protoA2 st m) protoC)
                 (initState: State) : Prop :=
  (x, st1) ← doProto (protoA1 initState) protoB;
  (y, st2) ← doProto (protoA2 st1 x) protoC;
  protoReturn (y, st2).
```

Figure 7.1. hypothetical protocol session composition

ity of protocol expressions encourages flexible protocol compositions that accept diverse protocols properly constrained by `Dual` for compatibility.

Verification of these protocol compositions presents some unique challanges. One contributor is that a protocol can *learn* messages in one protocol session, then use them in the next. For instance, if we would like to prove that a protocol cannot decrypt a message, we must prove that it cannot learn the decryption key *in any way* from its preceding sessions. Thus, we must have some sort of compositional verification strategy that incorporates `State`. One popular approach called *Hoare triples*[Hoare, 1969] decorates a stateful command with a *precondition* over the initial state and a *postcodition* over the terminal state. In Figure 7.2, we present a hypothetical Hoare triple style encoding in Coq for our protocol execution semantics. `P` and `Q` are `Assertions`–propositions over `State`–and they represent the precondition and postcondition, respectively. Pierce et al. [2015] confirm the feasibility of Hoare triple style verification in Coq over programming language expressions.

```
Definition Assertion := State → Prop.

Definition hoare_triple{p1t p2t p3t:protoType}{t:type}
               (P:Assertion)
               (p1:protoExp p1t) (p2:protoExp p2t) (p3:protoExp p3t)
               (Q:Assertion) : Prop :=
  ∀ st st',
     P st →
     multi st _ _ _ p1 p2 p3 st' →
     Q st'.
```

Figure 7.2. hypothetical Hoare triple framework for protocol execution

References

ArmoredSoftware. Armored software github repository, 2016. URL https://github.com/armoredsoftware.

J. Carette, W. Farmer, L. Cruz-Filipe, and P. Letouzey. Proceedings of the 12th symposium on the integration of symbolic computation and mechanized reasoning (calculemus 2005) a large-scale experiment in executing extracted programs. *Electronic Notes in Theoretical Computer Science*, 151(1):75 – 91, 2006. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2005.11.024. URL http://www.sciencedirect.com/science/article/pii/S1571066106001101.

A. Chlipala. *Certified programming with dependent types: a pragmatic introduction to the coq proof assistant*. MIT Press, 2013.

G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10(2):63–81, June 2011.

Coq development team. *Coq Reference Manual*. INRIA, 2016. URL https://coq.inria.fr/distrib/current/refman/. Version 8.5pl1.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation – a virtual machine directed approach to trusted computing. In *Proceedings of the Third Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.

C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–580,583, 1969.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, pages 122–138, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64302-8. URL http://dl.acm.org/citation.cfm?id=645392.651876.

X. Leroy. The compcert c verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, Nov. 1995. ISSN 0020-0190. doi: 10.1016/0020-0190(95)00144-2. URL http://dx.doi.org/10.1016/0020-0190(95)00144-2.

A. Martin et al. The ten page introduction to trusted computing. Technical Report CS-RR-08-11, Oxford University Computing Labratory, Oxford, UK, 2008.

R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978. ISSN 0001-0782. doi: 10.1145/359657.359659. URL http://doi.acm.org/10.1145/359657.359659.

K. Ono, Y. Hirai, Y. Tanabe, N. Noda, and M. Hagiya. Using coq in specification and program extraction of hadoop mapreduce applications. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, SEFM'11, pages 350–365, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24689-0. URL http://dl.acm.org/citation.cfm?id=2075679.2075705.

L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of computer security*, 6(1):85–128, 1998.

A. Petz. Github repository for protocol semantics and proofs in coq, August 2016. URL https://github.com/armoredsoftware/session/releases/tag/thesis_08_08_16.

B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2015.

R. Pucella and J. A. Tov. Haskell session types with (almost) no class. *SIGPLAN Not.*, 44(2):25–36, Sept. 2008. ISSN 0362-1340. doi: 10.1145/1543134.1411290. URL http://doi.acm.org/10.1145/1543134.1411290.

P. Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, Nov. 2015. ISSN 0001-0782. doi: 10.1145/2699407. URL http://doi.acm.org/10.1145/2699407.