Specifying Distributed Trust Management in LolliMon

Jeff Polakow
Harvey Mudd College
jpolakow@cs.hmc.edu

Christian Skalka
University of Vermont
skalka@cs.uvm.edu

Abstract

We propose the monadic linear logic programming language LolliMon as a new foundation for the specification of distributed trust management systems, particularly the RT framework. LolliMon possesses features that make it well-suited to this application, including rigorous logical foundations, an expressive formula language, strong typing, and saturation as a proof resolution strategy. We specify certificate chain discovery in full RT for authorization in a distributed environment where certificates may be stored non-locally and selective retrieval is necessary. The uniform LolliMon specification of authorization and certificate chain discovery eases formal reasoning about the system, and scales to a rich collection of trust management features. The executable LolliMon specification also serves as a prototype implementation.

Categories and Subject Descriptors C.2.0 [*Computer Networks*]: General—Security and protection

General Terms Security, Languages, Theory

Keywords Distributed Authorization, Trust Management Logic

1. Introduction

Distributed trust management supports resource protection in modern distributed computing environments. Trust management systems provide a framework to express security policies, provide actors with a means to establish trust relations across machine boundaries, and formalize the semantics of authorization. Trust management systems such as SPKI/SDSI [23] and RT [17] are especially suited to modern distributed computing environments, since they establish security in decentralized settings, where collaborations between actors are loose and dynamic.

Decentralization in trust management systems is partly obtained by using *certificates*, which are issued to individual entities in the system, allowing them to establish trust *credentials* in some particular domain, independent of a central authority. While authentication of certificates is a prerequisite for establishing credentials, authentication is orthogonal to authorization decisions, which are deductions based solely on credentials and local policy. But in a distributed environment, certificates to establish credentials relevant to any given decision may be spread across multiple machines, so authorization in these systems also requires retrieval of relevant certificates. The entire problem is referred to as *distributed certificate chain discovery* [18]. It is distinguished from the easier sub-problem of *chain discovery* [6, 14], in that the latter is only concerned with the implementation of authorization in a local credential environment, not with algorithms for interleaving non-local certificate retrieval and authorization steps.

PLAS'06 June 10, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-374-3/06/0006. . . $5.00.

1.1 Abstractions of Certificate Chain Discovery

Because a potentially enormous number of certificates may exist on disparate machines in a distributed environment, the trick is to be selective about retrieving certificates. One strategy is to let the frontiers of partially constructed authorization decisions inform the direction of retrieval. For example, trust relations can be formalized as edges in a graph, so that reachability implies authorization for a resource; hence, certificate chain discovery can be specified as a graph reconstruction algorithm, where future search proceeds from frontier nodes [18].

A drawback of the graph-theoretic abstraction of certificate chain discovery is that while it is adequate to represent simple trust relations, it is not scalable to more advanced features of trust management systems, e.g. delegation credentials, role attribute constraints, and threshold policies as in RT. Alternatively, programming logics such as Datalog or Prolog can be used to specify these features [18]. In a programming logic setting, credentials and policy are expressed as formulas, and authorization is specified as proof search. However, the restricted Horn-clause predicate languages of Datalog and Prolog prevent modeling the interleaving of retrieval and authorization phases of certificate chain discovery. Rather, these approaches treat only local authorization in a set of credential facts given *a priori*. Hence, certificate chain discovery in the full RT framework is an open problem.

While some scheme for re-compiling and re-running a Horn-clause logic program in a fact base extended by fresh certificate discovery would work, the approach is unappealing in several respects. For one, it does not provide a uniform model for rigorous verification of the algorithm. For another, it does not provide a natural means for memoizing partial solutions during discovery phases– since the proof search is restarted at each authorization phase, partial proofs constructed in previous phases are discarded.

1.2 A Logical Approach

The limitations of Prolog and Datalog do not necessarily undermine the usefulness of logic as an abstraction for RT certificate chain discovery. Rather, we argue that a programming logic with a more expressive formula language is needed to capture the logic of discovery, as well as the core authorization semantics. In this paper, we propose the use of the LolliMon linear logic programming language [19] as a new foundation for certificate chain discovery in RT. We show that LolliMon possesses a good mixture of features and proof strategies, including hypothetical goals, linear assumptions, and forward-chaining proof search, to allow clean integration of authorization checking and credential retrieval for certificate chain discovery. Furthermore, logic serves as an understandable specification of authorization semantics, and the rich formal theory underlying linear logic provides powerful tools for verifying properties of the model. While previous work has provided a graph-theoretic specification of certificate chain discovery for only the most basic variant of the RT framework [18], we believe our model is the first that scales to the full RT framework. In addition to a powerful specification language, using LolliMon also gives us a free prototype implementation which can serve as the basis for a future realistic implementation effort. Source code for the specification is available online [22].

1.3 Paper Outline

The remainder of the paper is organized as follows. In Sect. 2, we give a brief summary of RT and LolliMon. In Sect. 3, we define forward and backward chaining specifications of RT$_0$, which are proven equivalent. In Sect. 4, we show how LolliMon can be used to specify certificate chain discovery. These specifications are then extended to other RT variants in Sect. 5. We conclude with remarks on future work in Sect. 6.

2. Background: RT and LolliMon

In this section we provide a brief summary of the RT trust management system and the LolliMon language. Citations direct the reader to more detailed accounts in the literature.

2.1 The RT Framework

The RT trust management framework is thoroughly motivated and characterized by Li and Mitchell [17]. The framework is a family of languages, each of which is a variation on a core system called RT$_0$. In RT$_0$, individual actors, or principals, are called $\mathit{Entities}$ and are defined by public keys. We let A, B, C, D, E range over entities. Each entity A can create an arbitrary number of Roles in a namespace local to the entity, denoted $A.r$. The $\mathit{RoleExpressions}$ of RT$_0$, denoted f, are either entities or roles or constructed from other role expressions by *linking* and *intersection*, as described below. To define a role an entity issues credentials that specify the role's membership. Some of these credentials may be a part of private policy; others may be signed by the issuer and made publicly available. The overall membership of a role is taken as the memberships specified by all the defining credentials.

RT$_0$ provides four credential forms,or *types*:

Type 1. $A.r \longleftarrow E$

This form asserts that entity E is a member of role $A.r$.

Type 2. $A.r \longleftarrow B.s$

This form asserts that all members of role $B.s$ are members of role $A.r$. Credentials of this form can be used to delegate control over the membership of a role to another entity.

Type 3. $A.r \longleftarrow B.s.t$

This form asserts that for each member E of $B.s$, all members of role $E.t$ are members of role $A.r$. Credentials of this form allow linking to non-local namespaces; observe that E need not be known by A. The expression $B.s.t$ is called a *linked role*.

Type 4. $A.r \longleftarrow B_1.r_1 \cap \cdots \cap B_n.r_n$

This form asserts that each entity that is a member of all role expression forms $B_1.r_1, \ldots, B_n.r_n$ is also a member of role $A.r$. The expression $B_1.r_1 \cap \cdots \cap B_n.r_n$ is called an *intersection role*.

Authorization is then cast as a role membership decision: an access target is represented as some role $A.r$, and authorization for some entity B succeeds iff B is provably a member of $A.r$. In such a decision, we call $A.r$ the *governing role*. Authorization always assumes some given finite set of credentials, denoted $\mathcal{C}$. We use $\mathit{Entities}(\mathcal{C})$ to represent the entities used in a particular set of credentials $\mathcal{C}$, and similarly $\mathit{RoleNames}(\mathcal{C})$, $\mathit{Roles}(\mathcal{C})$, etc.

EXAMPLE 2.1. *Given the following credentials:*

$$A.r_1 \longleftarrow B.r_2.r_3 \cap C.r_4 \qquad B.r_2 \longleftarrow E \qquad E.r_3 \longleftarrow D$$
$$C.r_4 \longleftarrow E.r_3$$

Then, writing $A \in B.r$ to denote that A is a member of $B.r$, we can deduce:

$$E \in B.r_2 \qquad D \in E.r_3 \qquad D \in C.r_4 \qquad D \in A.r_1$$

2.2 Monadic Linear Logic Programming in LolliMon

LolliMon [19] is a new linear logic programming language, that cleanly combines backward chaining execution, aka top-down proof search, with forward chaining execution, aka bottom-up proof search. This integration is achieved via a monadic formula constructor which safely encapsulates forward chaining computations inside of backward chaining computations. In addition to a monad, LolliMon features typed, higher-order terms, and contains the full complement of intuitionistic linear logic connectives. The logic underlying LolliMon is based on the Concurrent Logical Framework (CLF) [24]. LolliMon's operational semantics (i.e. proof search strategy) and several interesting example programs are discussed in detail by López et al. [19]. For reference, Appendix 7 contains a complete presentation of the logic underlying LolliMon.

LolliMon has two main computation modes, backward chaining and forward chaining. Backward chaining computation is the standard Prolog operational semantics; proof search is directed by the shape of the goal, and atomic goals are analogous to function calls. Like Prolog, LolliMon's backward chaining proof search is depth first and subject to the usual looping behavior. Forward chaining computation, on the other hand, is similar to bottom-up logic programming semantics. Rather than being goal directed, the computation proceeds in a series of steps in which formulas are deduced from, and then added to, the current context until a fixed point, aka *saturation*, is reached (i.e. no change can be made in the context). LolliMon makes backward chaining the primary execution mode; every LolliMon execution starts, and ends, in backward chaining mode. The system switches to forward chaining mode upon encountering a monadic goal of the form $\{\mathtt{S}\}$, where $\mathtt{S}$ is a formula. After forward chaining finishes, the system reverts to backward chaining mode to solve goal $\mathtt{S}$.

The primary difference between linear logic programming and more standard logic programming is that the former does not allow weakening or contraction in proof contexts, as is the case for the unrestricted proof contexts of Prolog and Datalog. That is, in the spirit of linear logic [9], facts are like resources, that are consumed when used in the proof of a judgement– the same linear fact cannot be used more than once in the proof. LolliMon conservatively extends the language Lolli, and the reader is directed to Hodas and Miller [10] for background on the basics of linear logic programming.

In LolliMon, both linear and unrestricted proof contexts are available, as is unrestricted logical implication $\supset$. Linear connectives $\multimap$ and $\otimes$ can be thought of as linear analogues of unrestricted implication and conjunction $\wedge$, respectively. LolliMon uses the following concrete syntax:

$$\multimap \;=\; \texttt{-o} \qquad \circ\!\!- \;=\; \texttt{o-} \qquad \supset \;=\; \texttt{=>} \qquad \subset \;=\; \texttt{<=}$$
$$\otimes \;=\; \texttt{,} \qquad \top \;=\; \texttt{top}$$

Another distinction of LolliMon is that its predicate clauses are not restricted to a Horn clause form, but are more general linear logic formulas as defined in Appendix 7. In particular, this means that hypothetical goals `S => S` are allowed. In fact, the LolliMon form `(Q,R) => S` is syntactic sugar for `Q => R => S`. The relevance of this will be discussed in Sect. 4, and in general subtleties of the language will be discussed as they become relevant in the remaining text.

3. RT Authorization Semantics in LolliMon

In this section we give a LolliMon specification of RT_0. We begin with the encoding of credentials and policies, as well as a backward chaining definition of authorization that provides an intuitive specification of RT_0 semantics in familiar Horn-clause form. This specification also draws a clear connection with the original RT_0 semantics proposed by Li and Mitchell [17]. We observe shortcomings of backward chaining due to non-termination issues, and define a logically equivalent forward chaining specification that resolves these issues.

3.1 Credentials and Role Membership Encoding

We use the type language of LolliMon to specify the types of *entities*, *role names*, and *role expressions* within the logic:

```
entity : type        role_name : type

                role_expr : type
```

On this basis, entities, role, linked roles, and intersection roles are encoded by application of particular constructors (where `->` is the usual function type constructor):

```
^            : entity -> role_expr.
role         : entity -> role_name -> role_expr.
linked_role  : entity -> role_name ->
               role_name -> role_expr.
inter        : list role_expr -> role_expr.
```

RT entity expressions are then encoded by the function $[\![\cdot]\!]$ as follows, where $\hat{A}$ and $\hat{r}$ are the conventional encodings of entity and role names; we will generally just rewrite identifiers with all-lowercase ascii:

$$[\![A]\!] = (\texttt{\^{}}\ \hat{A}) \qquad [\![A.r]\!] = \texttt{role}\ \hat{A}\ \hat{r}$$

$$[\![A.r_1.r_2]\!] = \texttt{linked\_role}\ \hat{A}\ \hat{r}_1\ \hat{r}_2$$

$$[\![f_1 \cap \cdots \cap f_n]\!] = \texttt{inter}\ ([\![f_1]\!]\texttt{::}\cdots\texttt{::}[\![f_n]\!]\texttt{::nil})$$

The cons (`::`) and empty list (`nil`) constructors are provided in LolliMon for the built-in `list` datatype.

As for credentials, we depart from Li and Mitchell [16] where credentials are represented as Horn clauses with subgoals. Rather, we represent credentials in a knowledge base as atoms. As is shown in Sect. 4, we implement chain discovery via hypothetical subgoals, with retrieved credentials as the condition. The representation of credentials as atoms allows these hypotheses to be first-order, contributing to the simplicity of the specification and efficiency of the implementation. Thus:

```
credential : entity -> role_name -> role_expr -> o.
```

with the encoding extended to credentials as follows:

$$[\![A.r \longleftarrow f]\!] = \texttt{credential}\ \hat{A}\ \hat{r}\ [\![f]\!].$$

We let $[\![\mathcal{C}]\!]$ denote the obvious extension of $[\![\cdots]\!]$ to sets of credentials. In the type of `credential`, the symbol `o` represents the built-in LolliMon predicate type.

EXAMPLE 3.1. *Given:*

```
a : entity.     b : entity.     r0 : role_name.

      r1 : role_name.       r2 : role_name.
```

The linked role $A.r_1.r_2$ *is denoted by* `linked_role a r1 r2`*, and the credential* $B.r_0 \longleftarrow A.r_1.r_2$ *is represented as the atom:*

```
credential b r0 (linked_role a r1 r2).
```

Given these constructions, the predicate `ismem` is defined as follows. Other than the trivial modification necessary to treat credentials as atoms, this semantics is identical to that defined by Li and Mitchell [16].

```
ismem : role_expr -> entity -> o.

ismem (role A R) B <= credential A R (^ B).

ismem (role A R0) D <=
  credential A R0 (role B R1),
  ismem (role B R1) D.

ismem (role A R0) E <=
  credential A R0 (linked_role B R1 R2),
  ismem (role B R1) D,
  ismem (role D R2) E.

ismem (role A R) E <=
  credential A R (inter Res),
  ismems Res E.
```

The auxiliary predicate `ismems` iterates through the list of roles provided in an intersection role.

```
ismems : list role_expr -> entity -> o.

ismems nil B.

ismems ((role A R)::Res) B <=
  ismem (role A R) B,
  ismems Res B.
```

Role membership is then formally based on the `ismem` predicate, as follows.

DEFINITION 3.1. *Let* Σ *contain the specification of* `ismem` *above. Given credentials* $\mathcal{C}$*, an entity* A *is a member of a role* $B.r$ *iff* $\Sigma, [\![\mathcal{C}]\!]; \cdot \Rightarrow \texttt{ismem}\ [\![B.r]\!]\ \hat{A}$ *is derivable.*

3.2 The Forward Specification

A significant problem with the backward specification is that due to the top-down implementation of non-monadic formulae, cyclic credentials cause non-termination. For example, a credential set containing $A.r \longleftarrow B.r$ and $B.r \longleftarrow A.r$ could cause `ismem` to diverge. One approach to this problem would be to extend LolliMon with tabling, just as XSB [11] has been proposed as a foundation for SDSI/SPKI [14]. Instead, we exploit the monad in LolliMon to switch to a bottom-up proof search strategy, ensuring termination of our specification in the presence of cyclic constraints. To this end, we redefine `ismem` as follows (where `ismems` is unchanged from above).

```
credential A R (^ B) => {!ismem (role A R) B}.

credential A R0 (role B R1),
ismem (role B R1) D =>
{!ismem (role A R0) D}.

credential A R0 (linked_role B R1 R2),
ismem (role B R1) D,
ismem (role D R2) E =>
{!ismem (role A R0) E}.

credential A R (inter Res),
ismems Res B =>
{!ismem (role A R) B}.
```

Note the use of the unrestricted modality (`!`), allowing weakening and contraction over deduced `ismem` atoms; without it, deduced `ismem` atoms would be treated as linear atoms by default. This definition is logically equivalent to the backward specification, as we subsequently demonstrate; the only difference is that the heads of clauses are encapsulated within the monad, forcing the clauses to be used for forward chaining. Definition 3.1 is easily modified to accommodate this specification.

DEFINITION 3.2. *Let Σ' contain the specification of* `ismem` *above. Given credentials $\mathcal{C}$, an entity A is a member of a role $B.r$ iff $\Sigma', [\![\mathcal{C}]\!]; \cdot \Rightarrow \{\texttt{ismem}\ [\![B.r]\!]\ \hat{A}\}$ is derivable.*

3.3 The Proof Context as Partial Solution

As the proof process proceeds, forward chaining proof search will add `ismem` atoms to the proof context. In this way, the proof context maintains and extends a partial solution of the `ismem` predicate. An advantage of this implementation feature is that the context can be cached for reuse over multiple `ismem` queries so the same atoms need not be re-computed. For example, the query $\Sigma', [\![\mathcal{C}]\!]; \cdot \Rightarrow \{\texttt{ismem}\ [\![A.r]\!]\ \hat{B},\ \texttt{ismem}\ [\![C.r]\!]\ \hat{D}\}$ will compute the `ismem` atoms from $[\![\mathcal{C}]\!]$ once, and then check the two specific queries. Another advantage has to do with chain discovery, as will be discussed in Sect. 4.

EXAMPLE 3.2. *Given the definitions in Example 3.1, assume also the existence of the following entities and (cyclic) credentials:*

```
    c : entity.              d : entity.

credential a r1 (^ c).  credential a r2 (role c r2).

credential c r2 (^ d).  credential c r2 (role a r2).
```

Then the query $\Sigma', \Gamma; \cdot \Rightarrow \{\texttt{ismem (role a r2) d}\}$ will succeed, where Γ contains the preceding credentials, since the following unrestricted assertions will first be deduced by the forward chaining `ismem` *clauses:*

```
                ismem (role a r2) d.
  ismem (role c r2) d.     ismem (role a r1) c.
```

3.4 Equivalence of Specifications

We now demonstrate that the backward and forward chaining versions of `ismem` are logically equivalent. Subsequently, we will base our implementation of RT on the forward chaining version of `ismem`. This theorem establishes the core of correctness for our chain discovery technique with regard to the RT specification, and illustrates the tools for formal reasoning available in LolliMon. Proofs are given in the Appendix 8. We formally state equivalence of the two specifications as follows:

DEFINITION 3.3. *Let Σ' contain the monadic, forward chaining version of* `ismem`, *let Σ contain the non-monadic, backward chaining version of* `ismem`, *and assume Γ_C contains credential assertions, i.e. atoms of the form* `cred` $a\ r\ e$ *for some a, r, and e. Then equivalence of specifications is characterized by the relation:*

$$\Sigma', \Gamma_C; \cdot \Rightarrow \{\texttt{ismem (role A R) B}\}$$
iff
$$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismem (role A R) B}$$

We begin by showing the first direction of the equivalence, starting with a key lemma stating that a forward chaining derivation of `ismem` using the monadic specification implies the existence of a backwards chaining derivation using the original specification. The key to this lemma is the assumption that every `ismem` atom used by the forward chaining derivation is itself derivable with the usual specification.

LEMMA 3.1. *Let Γ_C contain credential assertions, let Γ contain* `ismem (role A R) B` *atoms, and assume that for all:*

$$(\texttt{ismem (role A' R') B'}) \in \Gamma$$

we have:

$$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismem (role A' R') B'}$$

Then the following properties hold:

1. *If* $\Sigma', \Gamma_C, \Gamma; \cdot \longrightarrow \texttt{ismem (role A R) B}$ *then* $\Sigma, \Gamma_C; \cdot \Rightarrow$ `ismem (role A R) B`
2. *If* $\Sigma', \Gamma_C, \Gamma; \cdot \Rightarrow \texttt{ismems Res B}$ *then* $\Sigma, \Gamma_C; \cdot \Rightarrow$ `ismems Res B`

We note that the Proof of part 1 does not rely on induction. In order to deal with the intersection case, we need to simultaneously prove that `ismems` can be derived by both specifications.

We may now state and directly prove the first part of the equivalence.

THEOREM 3.1. *Letting Γ_C contain credential assertions, if:*

$$\Sigma', \Gamma_C; \cdot \Rightarrow \{\texttt{ismem (role A R) B}\}$$

then also $\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismem (role A R) B}$.

Proof. By inversion on the given derivation and an appeal to Lemma 3.1.

We next proceed with the second part of our equivalence. Again we prove an auxiliary lemma that establishes the crux of the result. This lemma essentially shows that a backwards chaining derivation can be "substituted" for a hypothesis in a forward chaining derivation.

LEMMA 3.2. *Letting Γ_C contain credential assertions and Γ contain* `ismem (role A R) B` *atoms, if both of the following hold:*

$$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismem (role A R) B}$$

$$\Sigma', \Gamma_C, \Gamma, \texttt{ismem (role A R) B}; \cdot \longrightarrow S$$

then so does $\Sigma', \Gamma_C, \Gamma; \cdot \longrightarrow S$.

We may now show the second direction of the equivalence.

THEOREM 3.2. *Letting Γ_C contain credential assertions, if:*

$$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismem (role A R) B}$$

then also $\Sigma', \Gamma_C; \cdot \Rightarrow \{\texttt{ismem (role A R) B}\}$.

4. Distributed Certificate Chain Discovery

In a distributed setting, RT authorization for some resource might rely on a set of credentials, not all of which may be on hand. Any realistic implementation must provide not just a means for proving role membership based on a set of credentials, but also a means of deciding which certificates may be needed to complete authorization, for collecting them, and for integrating them as credentials into the proof procedure. Hence, our LolliMon specification must capture this extra functionality. It is hard to see how certificate retrieval phases could be integrated with role membership inferencing steps in Prolog or Datalog, due to their restricted formula languages [18]. However, the more expressive formula language of

LolliMon provides the necessary abstractions. Intuitively, our technique for interleaving certificate collection and inferencing will be achieved as follows: to prove an authorization goal, we must either prove membership via `ismem`, *or* show that the condition of additional certificate discovery entails authorization. The latter entailment is easily framed as a *conditional subgoal*.

4.1 Certificates as Linear Assertions

It is essential to keep in mind the distinction between certificates and credentials. The former, retrieved during discovery, are used to establish the latter for authorization inference steps. In particular, credentials may be used multiple times to establish a role membership, as the credential $E.r_3 \longleftarrow D$ must be used twice to establish that $D \in A.r_1$ in Example 2.1. In contrast, discovery of any particular certificate should occur only once, to ensure both efficiency and termination. We enforce this by modeling certificates as linear `entry` atoms. Since authorization uses the linear context, this means that any particular certificate entry can only be discovered once in an authorization proof. For example, the credential in Example 3.1 would be established by the following certificate:

```
#linear entry b r0 (linked_role a r1 r2).
```

While certificates would be stored on non-local machines in a distributed system, the details of retrieving non-local entries are abstracted in our model by linear hypothesis consumption[1].

As detailed in section 4.3, the task of determining which certificates to retrieve is handled by a predicate `seed`, which returns a credential to start a credential chain, and some forward chaining `credential` clauses which add new links to an existing credential chain. In this presentation, retrieval is determined by the entity being authorized and the governing role, as discussed below. The type signature and mode of the `seed` predicate are:

```
seed : role_expr -> entity ->
       entity -> role_name -> role_expr -> o.

   seed +Re +E -E' -Rn' -Re'.
```

The definition of `seed` will be specified later as part of the definitions of discovery schemes.

4.2 Certificate Chaining via Conditional Subgoals

In logic, the proof of an implication $A \Rightarrow B$ is obtained by assuming A as a fact, and then proving B. This well-known rule of inference can nicely specify the interleaving of retrieval and authorization in certificate chain discovery. That is, if A describes the condition of credential retrieval, and B describes the condition of successful authorization, then $A \Rightarrow B$ describes successful authorization after a credential retrieval. Guided by this insight, our specification of discovery leverages the ability to express hypothetical goals in LolliMon. Hypothetical goals allows new facts to be introduced into the proof environment as part of the logical specification. Not only does this allow for a faithful specification of discovery within the logic, but also a natural means to memoize partial authorization solutions between authorization and retrieval phases.

We define the authorization predicate `auth` as follows, such that `auth R A` succeeds if `ismem R A` holds, or if the condition of additional certificate retrieval allows successful authorization.

```
auth : role_expr -> entity -> o.

auth R A o- ismem R A, top.
auth R A o-
  seed R A B Rb RE,
  (credential B Rb RE => {auth R A}).
```

[1] Every successful `entry` subgoal consumes a linear `entry` and marks the retrieval of a remote entry.

Note that `seed` is given the role expressions involved in the authorization query, to determine the holder of the retrieved entry. The predicate `auth` is defined in terms of linear entailment `o-`, since discovery is predicated on linear assertions in the linear context. The first clause specifies that authorization succeeds if `ismem` succeeds in the current credential context. The final `top` subgoal is necessary to clean up any unused linear assumptions, i.e. not every distributed certificate need be retrieved. The second clause allows for the discovery of new credentials, and allows proof of authorization under the condition of newly discovered credentials, via the conditional subgoal `(credential B Rb RE => {auth R A})`. The postcondition is monadic, to eagerly drive forward-chaining inference in the implementation, e.g. for subsequent `ismem` subgoals, as well as discovery, as discussed below.

4.3 Directed Chain Discovery

Additional clauses for the predicate `credential` implement chain discovery. In effect, introducing a `credential` atom into the proof context via the conditional subgoal defined above “kick starts” the discovery process, with additional `credentials` added to the proof context by subsequent deductions.

The definition of `credential` is orthogonal with respect to the definition of authorization, so different discovery techniques can be used without any need to redefine authorization. However, since authorization integrates discovery, discovery of certificates is interleaved in the proof search. In effect, this means that authorization does not need to be “restarted” every time a new credential is discovered. As discussed in Sect. 3.3, this is because the proof context will maintain valid `ismem` assertions as they’re deduced during forward-chaining proof search, effectively memoizing the solution between discovery phases.

In chain discovery, efficiency is usually obtained by minimizing the number of credentials used to reconstruct a proof of authorization– the credential “chain” [6, 14]. In distributed chain discovery, selective use of credentials is even more important, since non-local certificate retrieval is computationally expensive, and the distributed environment can contain a potentially enormous number of them.

Another important factor in distributed chain discovery is the convention for credential storage, since this will determine how relevant credentials $A.r \longleftarrow f$ can be found. Li et al. [18] envision two scenarios. In one, credentials are stored with credential subjects– that is, entities B occurring in f. In another, credentials are stored with the issuer– that is, A. The original terminology refers to the former as forward chain discovery, and the latter as backward chain discovery, but to avoid confusion with LolliMon proof direction terminology, we instead call them *subject-driven* and *issuer-driven* discovery. In the remainder of this section we refigure the discovery techniques for each scenario in LolliMon. We note that as in Li at al.’s approach [18], subject- and issuer-driven techniques can be composed to obtain *bidirectional* discovery.

4.3.1 Subject-Driven Discovery

Given an authorization query of the form `auth` $[\![A.r]\!]\ \hat{B}$, subject-driven discovery starts by obtaining an entry of the form $C.s \longleftarrow B$, which by convention is held by B. Subsequent credential discovery is then driven by building a chain of credentials in the subject-to-issuer direction. Hence, we define `seed` as follows:

```
seed R A B Rb (^ A) o- entry B Rb (^ A).
```

Note that `seed` ignores `R` in this scheme, since certificates are retrieved in a subject-driven manner; and that successful retrieval consumes a linear certificate entry. We also define the following `credential` clauses. Note especially that entry retrieval is always determined by existing `credential` and `ismem` information, allowing selective certificate retrieval:

```
credential A Ra Re,
entry B Rb (role A Ra) -o
{!credential B Rb (role A Ra)}.

credential A Ra Re,
entry B Rb (inter Res),
subject Res A Ra -o
{!credential B Rb (inter Res)}.

credential A Ra Re,
entry B Rb (^ A) -o
{!credential B Rb (^ A)}.

credential A Ra Re,
ismem (role D R) A,
entry B Rb (linked_role D R Ra) -o
{!credential B Rb (linked_role D R Ra)}.
```

where (`subject` $[\![f_1 \cap \cdots \cap f_n]\!]$ $[\![A.r]\!]$) succeeds iff $\exists 0 < i \leq n.f_i = A.r$:

```
subject ((role A Ra)::Res) A Ra.
subject (X::Res) A Ra <= subject Res A Ra.
```

Proving correctness of credential chain discovery is a matter of relating the algorithm with the specification [18, 5]. To prove soundness of discovery, we demonstrate that successful chain discovery via `auth` entails valid role membership via `ismem`, where the latter is proved in an environment where `credentials` discovered during authorization are assumed (that is, localized). To this end, we make the following definition, which allows us to make a tight bound on the `entrys` retrieved during authorization. For the purposes of the definition, we make a trivial modification to `auth`: we remove `top` from the first clause and move it to queries, so that any authorization query is of the form `auth R A,top`.

DEFINITION 4.1. *Let Σ' be as given in Definition 3.3, let Γ_{disc} contain* `auth` *and* `credential` *clauses as defined above, and let Δ_{entry} contain linear* `entrys`*. Suppose the query* `auth R A,top` *is successful, i.e.* $\Sigma', \Gamma_{disc}; \Delta_{entry} \Rightarrow$ `auth R A,top` *is derivable. Then the* entries consumed by authorization *is the multiset Δ such that $\Delta_{entry} = \Delta, \Delta'$ and $\Sigma', \Gamma_{disc}; \Delta \Rightarrow$* `auth R A`.

Soundness can then be demonstrated as follows. Note that it is easy to show that any consumed `entry` will generate a corresponding `credential` that can be used in an `ismem` proof.

THEOREM 4.1. *Suppose $\Sigma', \Gamma_{disc}; \Delta_{entry} \Rightarrow$* `auth R A,top` *is derivable and Δ are the entries consumed by authorization, where $\Delta =$* `entry A1 R1 Re1,..., entry An Rn Ren`*. Let:*

$\Gamma_C =$ `credential A1 R1 Re1,...,credential An Rn Ren`

Then $\Sigma', \Gamma_C; \cdot \Rightarrow \{$`ismem R A`*$\}$ is derivable.*

In a similar vein, a completeness result can be formulated as follows.

THEOREM 4.2. *Suppose:*

$\Gamma_C =$ `credential A R1 Re1`$, \ldots,$ `credential A Rn Ren`

is the smallest credential set such that $\Sigma', \Gamma_C; \cdot \Rightarrow \{$`ismem R A`*$\}$ is derivable. Then $\Sigma', \Gamma_{disc}; \Delta \Rightarrow$* `auth R A` *is derivable, where $\Delta =$* `entry A R1 Re1`$, \ldots,$`entry A Rn Ren`.

4.3.2 Issuer-Driven Discovery

Issuer-driven discovery works under the assumption that credentials $A.r \longleftarrow f$ are stored with credential issuers. Hence, given an authorization query of the form `auth` $[\![A.r]\!]$ $\hat{B}$, issuer-driven discovery starts by obtaining an entry that defines the role $A.r$, which by convention are held by A. Subsequent credential discovery is then driven by building a chain of credentials in the issuer-to-subject direction. Hence, we define `seed` as follows:

```
seed (role A R) B A R Re o- entry A R Re.
```

Note that in this issuer-driven version of `seed` the subject `B` is ignored, in contrast to the subject-driven version in Sect. 4.3.1, where the issuer is ignored. We also define the following `credential` clauses. Observe that entry retrieval is always determined by existing `credential` and `ismem` information, allowing selective certificate retrieval:

```
credential A Ra (role B Rb),
entry B Rb RE -o
{!credential B Rb RE}.

credential A Ra (linked_role B Rb R2),
entry B Rb RE -o
{!credential B Rb RE}.

credential A Ra (linked_role B Rb Rc),
ismem (role B Rb) C,
entry C Rc RE -o
{!credential C Rc RE}.

credential A R (inter Res) -o
{!expand Res}.

expand (role A Ra::Res),
entry A Ra RE -o
{!credential A Ra RE, !expand Res}.
```

Correctness of issuer-driven discovery is proven in a manner similar to correctness of subject-driven discovery.

5. RT Framework Variations

In this section we specify both the authorization semantics and distributed chain discovery algorithms for variations on the basic RT system proposed by Li and Mitchell [17]. While the authorization semantics specifications are modeled on a previous Datalog specification of RT [17], the distributed certificate discovery specification is the first that applies to all variants in the RT framework.

5.1 Adding Constraints: RT$_1$ and RT$_2$

The systems RT$_1$ and RT$_2$ extend RT with *constrained role name parameters*. In RT$_1$, rather than being simply identifiers, role names can be parameterized by atomic data values such as integers and date/times, which can optionally be constrained to be in some range of values. Recalling Example 3 from Li and Mitchell [17], we could state the policy that "the founding alumni of State University include those who received a degree X in some year Y between 1955 and 1958" as follows:

$$\begin{array}{c} \mathit{StateU.foundingAlumni} \\ \longleftarrow \\ \mathit{StateU.diploma}(X, Y : [\mathit{1955..1958}]) \end{array} \qquad (1)$$

In RT$_2$, parameters may include entities optionally constrained to be in some role. Recalling example 4 from Li and Mitchell [17], we could state the policy that "Alpha company allows members of a project team to read documents of the project" as follows:

$$Alpha.file(read, X : Alpha.documents(Y)) \longleftarrow Alpha.team(Y) \quad (2)$$

We can specify this behavior in LolliMon by defining a new parameter constraint type, and modifying the type signature of credentials to include (possible empty) lists of constraints on parameters.

```
pconstraint : type.

credential  : entity -> role_name ->
              role_expr -> list pconstraint -> o.
```

We illustrate the encoding with three sorts of constraints– integer ranges, entity role membership, and a means of constraining an entity parameter to be the subject entity under consideration in a role membership decision (`this` in the original RT_1 terminology):

```
intc  : int -> int -> int -> pconstraint.
osetc : entity -> role_expr -> pconstraint.
thisc : entity -> pconstraint.
```

These constraints are endowed with a straightforward interpretation:

```
interp nil D.
interp (intc X B T::Ps) D <=
  leq B X,
  leq X T,
  interp Ps D.
interp (osetc E Re::Ps) D <=
  ismem Re E,
  interp Ps D.
interp (thisc D::Ps) D <= interp Ps D.
```

The forward-chaining LolliMon specification of `ismem` is then obtained by requiring credential constraints to be satisfied in the interpretation; given the obvious definition of `leq` as $\leq$ on integers[2], the linked role case would be defined as follows:

```
credential A R0 (linked_role B R1 R2) Cs,
ismem (role B R1) D,
ismem (role D R2) E,
interp Cs E =>
{!ismem (role A R0) E}.
```

And so on for the other forms.

LolliMon also has the benefit of a type system, that automatically enforces well-typedness of parameterized role names, unlike encodings in Datalog or Prolog. Also, variables can occur in arbitrary positions in LolliMon assertions, allowing formulae that would be "unsafe" and disallowed in Datalog. This in turn allows a first-order representation of credential discovery in the presence of constrained credential parameters, illustrated as follows. We illustrate this with the following example encoding, not safely expressible in Datalog.

EXAMPLE 5.1. *Credential (2) defined above can be encoded as follows:*

```
alpha : entity.
file_ac : entity -> entity -> role_name.
read : entity.
team : entity -> role_name.
documents : entity -> role_name.
credential
   alpha (file_ac read X) (role alpha (team Y))
   (osetc X (role alpha (documents Y))::nil).
```

[2] The current LolliMon prototype only provides $>$ and $=$.

Specifying certificate discovery for RT_1 is straightforward, essentially requiring a simple modification of `entrys` to accommodate constraints in the same manner as `credentials`. However, the `osetc` constraints that are characteristic of RT_2 pose a greater challenge. Firstly, any discovered `osetc` constraints must be treated as new authorization goals supported by their own credential chains. Thus we define a `cnstrDisc` predicate, that iterates through `pconstraint` lists and triggers credential retrieval for `osetc` constraints. The relevant clause is as follows:

```
cnstrDisc (osetc E Re::Cs) -o
{auth Re E, cnstrDisc Cs}.
```

Furthermore, note that in the Example 5.1 encoding of credential (2) the variables `X` and `Y` are universally quantified. When discovering this credential as part of a chain, the `credential` predicate will instantiate `X` and `Y`, but only to "fit it into the chain". Post-discovery, the credential should still be usable with full generality. Thus, we define a predicate `(test C D)` that only *tests* whether a credential `C` can be instantiated to a particular form `D`. These specifications are composed for type 2 credentials as follows:

```
credential A Ra (role B Rb) Cs1,
entry Cred,
test Cred (credential B Rb RE Cs2) -o
{!Cred, cnstrDisc Cs1}.
```

The remaining clauses are defined in a similar manner [22]. We observe that when solving `osetc` constraints, it is necessary to proceed in an issuer-driven manner, since the subject is arbitrary (e.g. `X` in Example 5.1).

5.2 Delegation: RT^D

The ability to Delegate rights from one entity to another is an important feature of trust management systems. In RT, this means the ability of an entity to delegate role membership "activations", which are weaker than role memberships. The system RT^D adds delegation credentials to the language, which are of the form:

$$B_1 \xrightarrow{D \text{ as } A.r} B_2$$

This says that B_1 has delegated its activation of the membership of D in $A.r$ to B_2. Intuitively, an entity can activate its own role memberships, or the activations that have been delegated to it by a delegation chain. A request for a resource *req* is encoded as a delegation of the desired role activation from the requester to the request. For example, if the *Registrar* has delegated $CompSci.Enroll$ membership to *Ryan* for the purpose of enrolling in CS courses, he can make a request for enrollment *enroll_req* by issuing:

$$Ryan \xrightarrow{Registrar \text{ as } CompSci.Enroll} enroll\_req \quad (1)$$

In LolliMon, we encode delegations via a predicate of the appropriate type:

```
delegation :
    entity -> entity -> role_expr -> entity -> o.
```

Validity of role activations is defined via a forward chaining predicate `for_role B D R` which holds iff `B` can activate the membership of `D` in `R`. Here we give some representative clauses; a complete definition would also include `for_role` clauses for type 3 (linked role) and type 4 (intersection role) credentials:

```
for_role : entity -> entity -> role_expr -> o.

delegation B1 D (role A R) B2,
for_role B1 D (role A R) =>
{!for_role B2 D (role A R)}.
```

```
credential A R (^ B) =>
{!for_role B B (role A R)}.

credential A R0 (role B R1),
for_role D E (role B R1) =>
{!for_role D E (role A R0)}.
```

The ability to activate one's own role membership is equivalent to role membership, hence:

```
ismem R A <= for_role A A R.
```

In RT^D, delegation credentials are assumed to be submitted along with a request for a resource, and so do not need to be retrieved. Therefore, the crux of authorization is to prove the underlying role membership for the desired activation, which can be established via previously discussed techniques. To wit:

```
auth : entity -> entity -> role_expr -> o.

auth A B R o- for_role A B R, top.
auth A B R o-
  seed R B D Rd RE,
  (credential D Rd RE => {auth A B R}).
```

where `seed` and `credential` are as defined in the subject-driven, issuer-driven, or bidirectional schemes previously defined in Sect. 4. Any authorization query can then be phrased as a hypothetical goal, where the preconditions are the delegation certificates issued along with the request.

EXAMPLE 5.2. *The example query expressed in delegation certificate (1) above can be formalized in LolliMon as follows. The membership of $Registrar$ in $CompSci.Enroll$ is established by the following certificate entry:*

```
#linear entry compsci enroll (^ registrar)
```

while the authorization query is a conditional goal, where the conditions are the request delegation, and the delegation of the Registrar's relevant role activation to Ryan:

```
delegation registrar registrar
           (role compsci enroll) ryan,
delegation ryan registrar
           (role compsci enroll) enroll_req
=> auth enroll_req registrar (role compsci enroll).
```

6. Conclusion

We now conclude with some remarks on related and future work.

6.1 Related Work

The LolliMon language, developed by López et al. [19], is based on ideas originally developed in the concurrent logical framework [24]; this work focused on the logic definition, not its application to trust management. Abadi [2] developed a semantics for SDSI based on propositional logic extended with axioms for SDSI namespaces. Similarly, Howell and Kotz [12] based a semantics for SPKI on the logic of authentication [1]. However, this work is only concerned with the authorization semantics SPKI/SDSI, not certificate retrieval. Clarke et al. [6] explored a local chain discovery problem for SPKI/SDSI, but it is based on a rewriting strategy, not logic programming. Li [14] proposed tabled logic programming, specifically XSB [11], as an alternate logical foundation for SDSI/SPKI, as well as an implementation language for chain discovery.

Li and Mitchell [17] used constraint Datalog as a logical foundation for specification of the RT framework, and studied the complexity of constraint Datalog in this application, as well as in its application to the KeyNote trust management system [4]. But like the logic-based characterizations of SPKI/SDSI discussed above, this work is focused on the semantics of authorization, not distributed chain discovery. Li et al. [18] addressed this latter problem for RT_0, but not in a logical framework, rather within an alternate set-theoretic semantics. Delegation Logic [15] is an application-specific logic for trust management, that provides a specification and implementation of a language with expressivity similar to RT. However, adaptation of their chain discovery technique to distributed settings is left as future work.

Koshutanski and Massacci [13] developed a framework for certificate chain discovery where authorization is characterized as deduction and certificate retrieval is characterized as abduction. In contrast to our approach, they propose a scheme whereby authorization can be stopped and restarted between abduction phases, and these may be implemented as completely separate components. They leave the actual definitions of deduction and abduction abstract. Their approach to certificate retrieval is based on application-specific sets of rules; unlike the approach discussed here, where future discovery is directed by partial authorization solutions, there is not necessarily a formal relation between the rules that specify abduction and policies for deduction. However, their approach to retrieval is promising and could augment our technique.

Proof carrying authorization (PCA) is a highly expressive distributed authorization system based on higher-order logic [3]. The system is implemented in the logical framework Twelf [21]. PCA is more powerful and complex than RT, with strong similarities to proof carrying code [20], and is intended for client-server interactions.

6.2 Future Work

An immediate direction for future work is to implement certificate chain discovery on the foundations described here. While the specification in this paper provides the first approximation of an implementation, since it is executable LolliMon code [22] and the forward chaining specification terminates, a number of issues remain. For example, a realistic architecture would require adopting a wire-format representation of entries, developing credential signing schemes, and defining and verifying protocols for query submission and credential retrieval.

A more theoretical direction for future work is an efficiency analysis of the LolliMon implementation of chain discovery. We believe we can develop complexity analysis techniques for LolliMon by building off of those for bottom-up logic programming [7, 8].

Another topic of interest is the efficiency and flexibility of retrieval techniques. Since credential retrieval requires network communication, it is a significant source of expense in the authorization procedure. The subject-driven, issuer-driven, and bidirectional discovery techniques discussed in Sect. 4 are not necessarily optimal. An abduction-based retrieval method [13] is markedly different, relying on rules provided in conjunction with authorization policies.

References

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

[2] Martin Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, 1998.

[3] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[4] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. *RFC-2704: The KeyNote Trust-Management System Version 2*. IETF, September 1999.

[5] Peter Chapin, Christian Skalka, and X. Sean Wang. Risk assessment in distributed authorization. In *Proceedings of the ACM Workshop on Formal Methods in Security Engineering*, 2005. To Appear.

[6] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[7] Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528. Springer, 2001.

[8] Harald Ganzinger and David A. McAllester. Logical algorithms. In Peter J. Stuckey, editor, *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 2002.

[9] J.-Y. Girard. Linear logic: Its syntax and semantics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic (Proc. of the Workshop on Linear Logic, Cornell University, June 1993)*. Cambridge University Press, 1995.

[10] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. In *Papers presented at the IEEE symposium on Logic in computer science*, pages 327–365, Orlando, FL, USA, 1994. Academic Press, Inc.

[11] XSB home page. `http://xsb.sourceforge.net`.

[12] Jon Howell and David Kotz. A formal semantics for SPKI. Technical Report 2000-363, Dartmouth College, 2000.

[13] H. Koshutanski and F. Massacci. An access control framework for business processes for web services. In *Proceedings of the ACM Workshop on XML Security*, 2003.

[14] Ninghui Li. Local names in SPKI/SDSI. In *PCSFW: Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[15] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security (TISSEC)*, February 2003. To appear.

[16] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages*, January 2003.

[17] Ninghui Li and John C. Mitchell. RT: A role-based trust-management framework. In *Proceedings of the Third DARPA Information Survivability Conference and Exposition*, pages 201–212. IEEE Computer Society Press, April 2003.

[18] Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, February 2003.

[19] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 35–46, New York, NY, USA, 2005. ACM Press.

[20] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[21] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.

[22] Jeff Polakow and Christian Skalka. A LolliMon specification of RT, 2006. `http://www.cs.uvm.edu/~skalka/lollimon/rt`.

[23] R. Rivest and B. Lampson. SDSI — a simple distributed security infrastructure, 1996. `http://citeseer.lcs.mit.edu/article/rivest96sdsi.html`.

[24] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, pages 355–377. Springer-Verlag LNCS 3085, 2004. Revised selected papers from the *Third International Workshop on Types for Proofs and Programs*, Torino, Italy, April 2003.

7. LolliMon Summary

This appendix contains a complete formal presentation of the logic underlying LolliMon.

FORMULA LANGUAGE:

$$\begin{array}{lcl} A & ::= & P \mid \top \mid A_1 \,\&\, A_2 \mid \\ & & A_1 \multimap A_2 \mid A_1 \supset A_2 \mid \forall x{:}\tau.A \mid \{S\} \\ S & ::= & A \mid !\,A \mid \mathbf{1} \mid S_1 \otimes S_2 \mid \exists x{:}\tau.S \end{array}$$

PROOF CONTEXTS:

$$\begin{array}{lcll} \Gamma & ::= & \cdot \mid \Gamma, A & \text{Unrestricted Context} \\ \Delta & ::= & \cdot \mid \Delta, A & \text{Linear Context} \\ \Psi & ::= & \cdot \mid S, \Psi & \text{Pattern Context} \end{array}$$

SEQUENT FORMS:

$$\begin{array}{rl} \Gamma; \Delta \Rightarrow A & \text{Right inversion} \\ \Gamma; \Delta; A \gg P & \text{Left focusing} \\ \Gamma; \Delta \gg S & \text{Right focusing} \\ \Gamma; \Delta \rightarrow S & \text{Forward chaining} \\ \Gamma; \Delta; A > S & \text{Monadic left focusing} \\ \Gamma; \Delta; \Psi \rightarrow S & \text{Left inversion} \end{array}$$

RIGHT INVERSION RULES:

$$\frac{\Gamma, A; \Delta; A \gg P}{\Gamma, A; \Delta \Rightarrow P}\text{uhyp} \qquad \frac{\Gamma; \Delta; A \gg P}{\Gamma; \Delta, A \Rightarrow P}\text{lhyp}$$

$$\frac{\Gamma; \Delta \rightarrow S}{\Gamma; \Delta \Rightarrow \{S\}}\{\}_R \qquad \frac{}{\Gamma; \Delta \Rightarrow \top}\top_R \qquad \frac{\Gamma, A; \Delta \Rightarrow B}{\Gamma; \Delta \Rightarrow A \supset B}\supset_R$$

$$\frac{\Gamma; \Delta, A \Rightarrow B}{\Gamma; \Delta \Rightarrow A \multimap B}\multimap_R \qquad \frac{\Gamma; \Delta \Rightarrow A \qquad \Gamma; \Delta \Rightarrow B}{\Gamma; \Delta \Rightarrow A \,\&\, B}\&_R$$

$$\frac{\Gamma; \Delta \Rightarrow [a/x]A}{\Gamma; \Delta \Rightarrow \forall x{:}\tau.A}\forall_R$$

LEFT FOCUSING RULES:

$$\frac{}{\Gamma; \cdot; P \gg P}\text{atm} \qquad \frac{\Gamma; \Delta; B \gg P \qquad \Gamma; \cdot \Rightarrow A}{\Gamma; \Delta; A \supset B \gg P}\supset_L$$

$$\frac{\Gamma; \Delta_1; B \gg P \qquad \Gamma; \Delta_2 \Rightarrow A}{\Gamma; \Delta_1, \Delta_2; A \multimap B \gg P}\multimap_L$$

$$\frac{\Gamma; \Delta; [t/x]A \gg P}{\Gamma; \Delta; \forall x{:}\tau.A \gg P}\forall_L \qquad \frac{\Gamma; \Delta; A \gg P}{\Gamma; \Delta; A \,\&\, B \gg P}\&_{L1}$$

$$\frac{\Gamma; \Delta; B \gg P}{\Gamma; \Delta; A \,\&\, B \gg P}\&_{L2}$$

FORWARD CHAINING RULES:

$$\frac{\Gamma, A; \Delta; A > S}{\Gamma, A; \Delta \rightarrow S}\text{uhyp}' \qquad \frac{\Gamma; \Delta; A > S}{\Gamma; \Delta, A \rightarrow S}\text{lhyp}'$$

$$\frac{\Gamma; \Delta \gg S}{\Gamma; \Delta \rightarrow S}\gg\rightarrow$$

MONADIC LEFT FOCUSING RULES:

$$\frac{\Gamma; \Delta; S' \to S}{\Gamma; \Delta; \{S'\} > S}\{\}_L \qquad \frac{\Gamma; \Delta; B > S \quad \Gamma; \cdot \Rightarrow A}{\Gamma; \Delta; A \supset B > S}\supset'_L$$

$$\frac{\Gamma; \Delta_1; B > S \quad \Gamma; \Delta_2 \Rightarrow A}{\Gamma; \Delta_1, \Delta_2; A \multimap B > S}\multimap'_L$$

$$\frac{\Gamma; \Delta; A > S}{\Gamma; \Delta; A \,\&\, B > S}\&'_{L1} \qquad \frac{\Gamma; \Delta; B > S}{\Gamma; \Delta; A \,\&\, B > S}\&'_{L2}$$

$$\frac{\Gamma; \Delta; [t/x]A > S}{\Gamma; \Delta; \forall x{:}\tau.A > S}\forall'_L$$

LEFT INVERSION RULES:

$$\frac{\Gamma; \Delta, A; \Psi \to S}{\Gamma; \Delta; A, \Psi \to S}\text{async} \qquad \frac{\Gamma; \Delta \to S}{\Gamma; \Delta; \cdot \to S}\to\to$$

$$\frac{\Gamma; \Delta; \Psi \to S}{\Gamma; \Delta; \mathbf{1}, \Psi \to S}\mathbf{1}_L \qquad \frac{\Gamma; \Delta; S_1, S_2, \Psi \to S}{\Gamma; \Delta; S_1 \otimes S_2, \Psi \to S}\otimes_L$$

$$\frac{\Gamma; \Delta; [a/x]S', \Psi \to S}{\Gamma; \Delta; \exists x{:}\tau.S', \Psi \to S}\exists_L \qquad \frac{\Gamma, A; \Delta; \Psi \to S}{\Gamma; \Delta; !\,A, \Psi \to S}!_L$$

RIGHT FOCUSING RULES:

$$\frac{\Gamma; \Delta \Rightarrow A}{\Gamma; \Delta \gg A}\Rightarrow\gg \qquad \frac{\Gamma; \cdot \Rightarrow A}{\Gamma; \cdot \gg !\,A}!_R \qquad \frac{}{\Gamma; \cdot \gg \mathbf{1}}\mathbf{1}_R$$

$$\frac{\Gamma; \Delta_1 \gg S_1 \quad \Gamma; \Delta_2 \gg S_2}{\Gamma; \Delta_1, \Delta_2 \gg S_1 \otimes S_2}\otimes_R \qquad \frac{\Gamma; \Delta \gg [t/x]S}{\Gamma; \Delta \gg \exists x{:}\tau.S}\exists_R$$

8. Equivalence of Specifications

Here we give proofs for various results stated in Sect. 3.4. We first establish some basic properties about LolliMon proofs which will be useful in the succeeding proofs.

LEMMA 8.1.

1. *If* $\Gamma, P; \cdot \to S$ *then* $\Gamma; \cdot; \{!P\} > S$.
2. $\Gamma, P; \cdot \Rightarrow P$
3. $\Gamma, P; \cdot \to P$

We now proceed to the proofs for Sect. 3.4.

PROOF OF LEMMA 3.1.

Part 1. By inspection of the given derivation making use of the assumption on Γ. There are exactly 5 cases to consider, one for each clause in Σ' plus one more for the case where Γ already contains the conclusion.

case: $\Sigma', \Gamma_C, \Gamma; \cdot; \texttt{ismem'\_role} > \texttt{ismem (role A R) B}$
$\Sigma', \Gamma_C, \Gamma; \cdot \Rightarrow \texttt{cred A R (role A' R')}$ and also $\Sigma', \Gamma_C, \Gamma; \cdot \Rightarrow \texttt{ismem (role A' R') B}$ by inversion,
$\texttt{cred A R (role A' R')} \in \Gamma_C$ by inversion,
$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{cred A R (role A' R')}$ by lemma 8.1,
$\texttt{ismem (role A' R') B} \in \Gamma$ by inversion (note clauses in Σ' are all monadic and not applicable),
$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismem (role A' R') B}$ by assumption,
$\Sigma, \Gamma_C; \cdot; \texttt{ismem\_role} \gg \texttt{ismem (role A R) B}$ by $\supset_L$,
$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismem (role A R) B}$ by uhyp
where

```
ismem_role =  ismem (role A' R') B ⊃
              cred A R (role A' R') ⊃
              ismem (role A R) B

ismem'_role = ismem (role A' R') B ⊃
              cred A R (role A' R') ⊃
              {!ismem (role A R) B}
```

Part 2. By structural induction on the given derivation making use of the assumption on Γ.

case: $\Sigma', \Gamma_C, \Gamma; \cdot; \texttt{ismems\_cons} \gg \texttt{ismems (role A R::Res) B}$
$\Sigma', \Gamma_C, \Gamma; \cdot \Rightarrow \texttt{ismem (role A R) B}$ and $\Sigma', \Gamma_C, \Gamma; \cdot \Rightarrow \texttt{ismems Res B}$ by inversion,
$\texttt{ismem (role A R) B} \in \Gamma$ by inversion (note clauses in Σ' are all monadic and not applicable),
$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismem (role A R) B}$ by assumption,
$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismems Res B}$ by induction hypothesis
$\Sigma, \Gamma_C; \cdot; \texttt{ismems\_cons} \gg \texttt{ismems (role A R::Res) B}$ by $\supset_L$,
$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismems (role A R::Res) B}$ by uhyp
where

```
ismems_cons = ismem (role A R) B ⊃
              ismems Res B ⊃
              ismems (role A R::Res) B
```

□

PROOF OF THEOREM 3.1. By inversion on the given derivation and an appeal to lemma 3.1. □

PROOF OF LEMMA 3.2. By structural induction on the first given derivation. There are 4 cases to consider, one for each clause in Σ.

case: $\Sigma, \Gamma_C; \cdot; \texttt{ismem\_role} \gg \texttt{ismem (role A R) B}$ and $\Sigma', \Gamma_C, \Gamma, \texttt{ismem (role A R) B}; \cdot \to S$
then
$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{cred A R (role A' R')}$ and $\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismem (role A' R') B}$ by inversion,
$\texttt{cred A R (role A' R')} \in \Gamma_C$ by inversion,
$\Sigma', \Gamma_C, \Gamma; \cdot \Rightarrow \texttt{cred A R (role A' R')}$ by lemma 8.1,
$\Sigma', \Gamma_C, \Gamma; \cdot; \{\texttt{!ismem (role A R) B}\} > S$ by lemma 8.1,
and letting $A = \{\texttt{!ismem (role A R) B}\}$ we have
$\Sigma', \Gamma_C, \Gamma, \texttt{ismem (role A' R') B}; \cdot; A > S$ by weakening,
$\Sigma', \Gamma_C, \Gamma, \texttt{ismem (role A' R') B}; \cdot; \texttt{ismem'\_role} > S$ by $\supset'_L$,
$\Sigma', \Gamma_C, \Gamma, \texttt{ismem (role A' R') B}; \cdot \to S$ by uhyp',
$\Sigma', \Gamma_C, \Gamma; \cdot \to S$ by induction hypothesis
where

```
ismem_role =  ismem (role A' R') B ⊃
              cred A R (role A' R') ⊃
              ismem (role A R) B

ismem'_role = ismem (role A' R') B ⊃
              cred A R (role A' R') ⊃
              {!ismem (role A R) B}
```

Note that the intersection case is just a generalization of the above case where all the roles in the intersection are weakened into the forward chaining hypotheses at once. □

PROOF OF THEOREM 3.2. Direct from lemma 8.1 and lemma 3.2 as follows:

$\Sigma, \Gamma_C; \cdot \Rightarrow \texttt{ismem (role A R) B}$ by assumption.
$\Sigma', \Gamma_C, \texttt{ismem (role A R) B}; \cdot \to \texttt{ismem (role A R) B}$ by lemma 8.1.
$\Sigma', \Gamma_C; \cdot \to \texttt{ismem (role A R) B}$ by lemma 3.2.
$\Sigma', \Gamma_C; \cdot \Rightarrow \{\texttt{ismem (role A R) B}\}$ by $\{\}_R$.

□