We propose the monadic linear logical programming language LolliMon as a new foundation for the specification and implementation of distributed trust management systems, particularly the RT framework. LolliMon possesses features that make it well-suited to this application, including rigorous logical foundations, hypothetical subgoals, linear assertions, strong typing, and saturation as a proof resolution strategy. These features allow for reliable, decidable, distributed authorization, where credentials may be stored non-locally and selective retrieval is necessary. The uniform presentation of RT in LolliMon allows for an easy proof of correctness for the implementation, and easy extension to a rich collection of trust management idioms.

1. INTRODUCTION

Distributed trust management supports resource protection in modern distributed computing environments. Trust management systems provide a framework to express security policies, and provide actors with a means to establish trust relations across machine boundaries. Trust management systems such as SPKI/SDSI [18] and RT [13] are especially suited to modern distributed computing environments, since they establish security in decentralized settings, where collaborations between actors are loose and dynamic. This paper focuses on logical foundations for trust management system specification and implementation, particularly within the RT framework.

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 within the framework, independent of a central authority. Authorization decisions are based on these certificates. However, because authorization policies can be complex, certificates relevant to any given decision may be spread across multiple machines. Thus, authorization in these systems is not just a matter of checking a given set of certificates for adherence to some policy, but also requires selective retrieval of relevant certificates. The entire problem is referred to as distributed certificate chain discovery [14]. It is distinguished from the similar problem of chain discovery [6; 10], in that the latter is only concerned with tractable authorization in a given credential environment, not with algorithms for interleaving credential retrieval with authorization steps.

Since security is important to get right, mathematically rigorous logics have been used as a linguistic foundation for trust management systems. These include logics specifically designed for application to trust management, such as the logic of authentication [1], as well as more general-purpose logics such as Datalog and Prolog. The former usually serve to specify the semantics of trust management systems, while the latter, being computational, also provide a means to implement localized authorization decisions—indeed, the specification is the implementation. However, the use of computational logics for implementation of distributed chain discovery has not been as well explored. This is unfortunate, since logics establish rigorous semantic foundations, and a uniform setting for specification and implementation of chain discovery would significantly ease proof effort, and raise confidence in security mechanisms. In this paper, we propose the use of the LolliMon linear logic programming language [15] as a new foundation for RT trust management, both for specification of credential semantics, and for implementation of distributed chain discovery. As we discuss below, LolliMon has a number of unique features that are well-suited to the task.
1.1 Related Work

The LolliMon language is developed in [15], which is based on ideas originally developed in the concurrent logical framework (CLF) [19]. These papers are focused on the logic definition, not its application to trust management. In [2] a semantics of SDSI is developed based on propositional logic extended with axioms for SDSI namespaces. Similarly, in [9] a semantics for SPKI is developed in a logic based on the logic of authentication [1]. However, this work is only concerned with the semantic specification of SPKI/SDSI, not the implementation. A chain discovery problem for SPKI/SDSI is explored in [6], but it is based on a rewriting strategy, not logic programming. In [10], Prolog with XSB is proposed as an alternate logical foundation for SDSI/SPKI, as well as an implementation language for chain discovery.

In [13], constraint Datalog is used as a logical foundation for specification of the RT framework. The complexity of constraint Datalog in this application, as well as in application to the KeyNote trust management system [4], is studied in [12]. But like the logic-based characterizations of SPKI/SDSI discussed above, this work is focused on semantic specification, not distributed chain discovery. This latter problem is addressed for RT$_0$ in [14], but not in a logical framework, rather within an alternate set-theoretic semantics. Delegation Logic [11] is an application-specific logic for trust management, that provides a specification and implementation of a language with expressivity similar to RT. However, an adaption of their chain discovery technique to a distributed setting is left as future work.

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 [17]. PCA is more powerful and complex than RT, with strong similarities to proof carrying code [16], and is intended for client-server interactions.

1.2 Contributions

This paper contributes to research in distributed trust management in two dimensions, by providing a more complete implementation of the RT framework, and by illustrating the effectiveness of an expressive logic programming language as a uniform foundation for the specification and implementation of trust management systems.

We show how the LolliMon language can be used to implement distributed chain discovery algorithms for the entire RT framework (modulo the mechanism for physical retrieval of non-local certificates, which aspect we model within the logic). Previous work [14] has only provided an implementation of distributed chain discovery of the most basic member of the RT family, called RT$_0$. We also show that this implementation is correct with respect to the original logical semantics of RT presented in [12]. The algorithm in [14] was proven correct only with respect to an alternate set-theoretic semantics, which is arguably less understandable than the logical semantics and has not been proven equivalent. Furthermore, it is difficult to see how the set-theoretic semantics could be extended to full RT, how one would extend the implementation, and how difficult it would be to prove the latter correct. Our approach avoids these issues, by founding both the specification and implementation of the system in the same logic programming language. This uniform setting significantly eases proof effort to demonstrate soundness of the implementation.

We use the LolliMon language, an intuitionistic linear logic programming language. The language has a number of features that make it useful for the specification and implementation of RT, with advantages over Datalog and Prolog in this application. In particular, LolliMon has a more expressive formula language, including hypothetical goals and linear assumptions, that allow clean integration of authorization checking and credential retrieval for distributed chain discovery as discussed in Sect. 4. Other benefits of LolliMon include the ability to mix top-down and bottom-up proof search strategies, and the ability to express formulae that would be “unsafe” in Datalog, as discussed in Sect. 3 and Sect. 5. Finally, LolliMon is well-typed, hence a number of informal RT credential typing disciplines prescribed in [12] become formal and automatically enforced within the LolliMon definition.
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 implement distributed chain discovery. These specifications and implementations 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 in [13]. 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 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 RoleExpressions of RT$^R$, 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:

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

2. $A.r \leftarrow B.s$
   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.

3. $A.r \leftarrow B.s.t$
   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 can be used to delegate control over the membership of a role to all entities that have the attribute represented by $B.s$. The expression $B.s.t$ is called a linked role.

4. $A.r \leftarrow B_1.r_1 \cap \cdots \cap B_n.r_n$
   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 $C$. We use $\operatorname{Entities}(C)$ to represent the entities used in a particular set of credentials $C$, and similarly $\operatorname{RoleNames}(C)$, $\operatorname{Roles}(C)$, etc.

Example 2.1. Suppose a hotel $H$ offers a room discount to certain preferred customers, who are members of $H.preferred$. The policy of $H$ is to grant a discount to all of its preferred customers in $H.preferred$ as well as to members of certain organizations. $H$ defines a role $H.orgs$ that contains the public keys of these organizations. Into that role $H$ places, for example, the key of the AAA, the American Auto Association. These credentials are summarized as follows:

\[
H.discount \leftarrow H.preferred \quad H.discount \leftarrow H.orgs.members \quad H.orgs \leftarrow AAA
\]

2.2 Monadic Linear Logic Programming in LolliMon

LolliMon [15] 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.
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) [19]. LolliMon’s operational semantics (i.e. proof search strategy) and several interesting example programs are discussed in detail in [15]. For reference, Appendix A contains a complete presentation of the logic underlying LolliMon. LolliMon is based on the language Lolli, and the reader is directed to [8] for background on the basics of linear logic programming.

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 is reached (i.e. no change can be made in the context). LolliMon makes backward chaining the primary execution mode. Thus 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 \{S\}, where \(S\) is a formula. After forward chaining finishes, the system reverts to backward chaining mode to solve goal \(S\).

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 [7], 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. In LolliMon, both linear and unrestricted proof contexts are available, as is unrestricted logical implication \(\top\). Linear connectives \(-\circ\) and \(\otimes\) can be thought of as linear analogues of unrestricted implication and conjunction \(\land\), respectively. LolliMon uses the following concrete syntax:

\[
- \circ = -o \quad o- = o- \quad \supset = => \quad \subseteq = <= \quad \otimes = , \quad T = top
\]

Another distinction of LolliMon is that its predicate clauses are not restricted to a Horn clause form, but are more general linear logic formulae as defined in the Appendix. In particular, this means that hypothetical goals \(S \Rightarrow S\) can be constructed. In fact, the LolliMon form \((Q,R) \Rightarrow S\) is syntactic sugar for \(Q \Rightarrow R \Rightarrow S\). 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. BACKWARD AND FORWARD CHAINING RT SEMANTICS

In this section we give a LolliMon specification of \(RT_0\). We begin with a backward chaining specification, and then define a forward chaining specification, with the latter resolving shortcomings of the former. We also demonstrate equivalence of the specifications.

3.1 The Backward Specification

In [13], a Datalog semantics for \(RT_0\) is defined. We now define an equivalent semantics in LolliMon. In addition to serving as a specification for the implementation of the system, it will serve as a starting point for the implementation itself. We will show that the specification is preserved by chain discovery, establishing a stronger result with respect to the original \(RT_0\) semantics of [13], since the algorithm in [14] is only proven correct with respect to an alternate set-theoretic semantics, that has not been formally related to the Datalog semantics. An advantage of our approach, is that extensions to \(RT_0\) will be easily treated by appropriate extensions to the specification, which are automatically carried through to the implementation.

To begin, we specify the types of entities, role names, and role expressions:

\[
\text{entity : type} \quad \text{role_name : type} \quad \text{role_expr : type}
\]
We then provide role expression constructors for entities, role, linked roles, and intersection roles respectively, where -> is a function type constructor:

\[\cdot : \text{entity} \to \text{role_expr}.\]
\[\text{role} : \text{entity} \to \text{role_name} \to \text{role_expr}.\]
\[\text{linked_role} : \text{entity} \to \text{role_name} \to \text{role_name} \to \text{role_expr}.\]
\[\text{inter} : \text{list role_expr} \to \text{role_expr}.\]

RT entity expressions are then encoded by the function \(\llbracket\cdot\rrbracket\) 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:

\[
\llbracket A \rrbracket = \hat{A}
\llbracket A.r \rrbracket = \text{role} \hat{A} \hat{r}
\llbracket A.r_1.r_2 \rrbracket = \text{linked_role} \hat{A} \hat{r}_1 \hat{r}_2
\llbracket f_1 \cap \cdots \cap f_n \rrbracket = \text{inter} (\llbracket f_1 \rrbracket ; \cdots ; \llbracket f_n \rrbracket ; \text{nil})
\]

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

As for credentials, we depart from [12] 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 this hypothesis to be first-order, contributing to the simplicity of the specification and efficiency of the implementation. Thus:

\[\text{credential} : \text{entity} \to \text{role_name} \to \text{role_expr} \to \text{o}.\]

with the encoding extended to credentials as follows:

\[
\llbracket A.r \leftarrow f \rrbracket = \text{credential} \ a \ r \ \llbracket f \rrbracket.
\]

We let \(\llbracket C \rrbracket\) denote the obvious extension of \(\llbracket \cdot \rrbracket\) to sets of credentials. In the type of \text{credential}, the symbol o represents the built-in LolliMon predicate type.

**Example 3.1.** Given:

\[a : \text{entity.} \quad b : \text{entity.} \quad r_0 : \text{role_name.} \quad r_1 : \text{role_name.} \quad r_2 : \text{role_name.}\]

The linked role \(A.r_1.r_2\) can be represented as \text{linked_role} \(a \ r_1 \ r_2\), and the credential \(B.r_0 \leftarrow A.r_1.r_2\) can be represented as the atom:

\[\text{credential} \ b \ r_0 \ (\text{linked_role} \ a \ r_1 \ r_2).\]

Given these constructions, the predicate \text{ismem} is defined as follows. The auxiliary predicate \text{issem} iterates through the list of roles provided in an intersection role. Other than the trivial modification necessary to treat credentials as atoms, this semantics is identical to that defined in [12].

\[\text{ismem} : \text{role_expr} \to \text{entity} \to \text{o}.\]
\[\text{issem} : \text{list role_expr} \to \text{entity} \to \text{o}.\]

\[\text{ismem (role A R) B} \leq \text{credential} \ A \ R \ (\sim B).\]
\[\text{issem (role A R0) D} \leq \text{credential} \ A \ R0 \ (\text{role B R1}),\]
\[\text{ismem (role B R1) D}.\]
\[\text{issem (role A R0) E} \leq \text{credential} \ A \ R0 \ (\text{linked_role} \ B \ R1 \ R2),\]
\[\text{issem (role B R1) D},\]
\[\text{ismem (role D R2) E}.\]
Role membership is then formally based on the `ismem` predicate, as follows.

**Definition 3.1.** Let $\Sigma$ contain the specification of `ismem` above. Given credentials $C$, an entity $A$ is a member of a role $B.r$ iff $\Sigma, [C]:: \Rightarrow \text{ismem} [B.r] A$ 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 formulæ, cyclic credentials cause non-termination. For example, a credential set containing $A.r \leftarrow B.r$ and $B.r \leftarrow A.r$ could cause `ismem` to diverge. One approach to this problem would be to extend LolliMon with tabling, as XSB-extended Prolog is proposed as a foundation for SDSI/SPKI in [10]. However, we instead exploit the monad in LolliMon to switch to a bottom-up proof search strategy, ensuring termination in the presence of cyclic constraints. We are thus able to reuse the existing LolliMon proof engine, without the addition of a complicated tabling mechanism. To this end, we redefine `ismem` as follows. 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. 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.

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

Definition 3.1 is easily modified to accommodate this specification.

### 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 authorization sessions, so the same atoms need not be re-computed. 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:

\[ c : \text{entity} \quad d : \text{entity} \quad \text{credential } c \text{ r2} (^d) \quad \text{credential } a \text{ r1} (^c) \quad \text{credential } a \text{ r2} (\text{role } c \text{ r2}) \quad \text{credential } c \text{ r2} (\text{role } a \text{ r2}) \]

Then the query \{ \text{ismem (role } b \text{ r0} ) d \} will succeed in a proof context containing the following unrestricted assertions:

\[ \text{ismem (role } c \text{ r2) d} \quad \text{ismem (role } a \text{ r1) c} \quad \text{ismem (role } b \text{ r0) d} \]
\[ \text{ismem (role } a \text{ r2) d} \]

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. Proofs are given in the Appendix. We formally state equivalence of the two specifications as follows:

**Definition 3.2.** Let \( \Sigma' \) contain the monadic, forward chaining version of ismem, let \( \Sigma \) contain the non-monadic, backward chaining version of ismem, and assume \( \Gamma_C \) containing credential assertions, i.e. atoms of the form \( \text{cred a r} (e) \) for some \( a, r, e \). Then equivalence of specifications is characterized by the relation:

\[ \Sigma', \Gamma_C ; \cdot \Rightarrow \{ \text{ismem (role } A \text{ R) B} \} \quad \text{if and only if} \quad \Sigma, \Gamma_C ; \cdot \Rightarrow \text{ismem (role } A \text{ R) B} \]

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. 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.

**Lemma 3.1.** Let \( \Gamma_C \) contain credential assertions, let \( \Gamma \) contain ismem (role A R) B atoms, and assume \( \forall (\text{ismem (role } A' \text{ R') B'} \in \Gamma) \). Then the following properties hold:

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

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

**Theorem 3.1.** Letting \( \Gamma_C \) contain credential assertions, if \( \Sigma', \Gamma_C ; \cdot \Rightarrow \{ \text{ismem (role } A \text{ R) B} \} \) then \( \Sigma, \Gamma_C ; \cdot \Rightarrow \text{ismem (role } A \text{ R) B} \).

**Proof.** By inversion on the given derivation and an appeal to Lemma 3.1. \( \square \)

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 \( \Gamma_C \) contain credential assertions and \( \Gamma \) contain ismem (role A R) B atoms, if \( \Sigma, \Gamma_C ; \cdot \Rightarrow \text{ismem (role } A \text{ R) B} \) and \( \Sigma', \Gamma_C, \Gamma, \text{ismem (role } A \text{ R) B} ; \cdot \Rightarrow S \) then \( \Sigma', \Gamma_C, \Gamma ; \cdot \Rightarrow S \).

We may now show the second direction of the equivalence.

**Theorem 3.2.** Letting \( \Gamma_C \) contain credential assertions, if \( \Sigma, \Gamma_C ; \cdot \Rightarrow \text{ismem (role } A \text{ R) B} \) then \( \Sigma', \Gamma_C ; \cdot \Rightarrow \{ \text{ismem (role } A \text{ R) B} \} \).
4. DISTRIBUTED CREDENTIAL CHAIN DISCOVERY

4.1 Credential Chaining as Proof of Conditional Subgoals

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 therefore provide not just an efficient means for proving role membership based on a set of credentials, but also a means for collecting new credentials and integrating them into the proof procedure. As observed in [14], it is hard to see how credential retrieval phases could be integrated with role membership inferencing steps in Prolog or Datalog. In LolliMon, however, the ability to express conditional subgoals presents a natural technique for interleaving credential collection and inferencing steps in a logic programming language. Intuitively, our technique for interleaving credential 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 an additional credential discovery entails authorization. The latter entailment is easily framed as a conditional subgoal.

It is essential to keep in mind the distinction between discovery and inference. While a given credential assertion may be used multiple times in a proof of some `ismem` assertion (due to linked roles), discovery of any particular credential should occur only once. We enforce this by modeling distributed credential entries as linear assertions. Since authorization uses the restricted context, this means that any particular entry can only be discovered once in an authorization proof. For example, the credential in Example 3.1 would be stored as an assertion:

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

While entries would be stored on non-local machines in a distributed system, the details of retrieving non-local entries are abstracted in this presentation. As we will see, entry lookup is determined by the entity we assume to be the “holder” of the entry. In a distributed system, this entity would have physical possession of the entry, so the only additional functionality needed for a truly distributed implementation of our technique would be a method of non-local entry retrieval from specified entities.

Putting these ideas together, and assuming the existence of a predicate:

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

that recovers stored entries, we can define the distributed authorization predicate `auth` as follows:

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

Note that `retrieve` is give 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 restricted 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 credential must 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.

4.2 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 in forward-chaining proof search for ismem assertions will maintain a list of valid assertions as they’re deduced, 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; 10]. In distributed chain discovery, selective use of credentials is even more important, since non-local credential 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 \( A.r \leftarrow f \) can be found. In [14], two scenarios are envisioned. 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.

### 4.2.1 Subject-Driven Discovery

Given an authorization query of the form \( \text{auth} \llbracket [A.r] \rrbracket \hat{B} \), forward chain discovery starts by obtaining all entries of the form \( A.r \leftarrow B \), which by convention are held by \( B \). Subsequent credential discovery is then driven by rebuilding a chain of credentials in the subject-to-issuer direction. Hence, we define retrieve as follows:

\[
\text{retrieve } R A B Rb (\neg A) \Leftarrow \text{entry } B Rb (\neg A) .
\]

and we define the following credential clauses, where \( (\text{subject } [f_1 \cap \cdots \cap f_n][A.r]) \) succeeds iff \( \exists 0 < i \leq n.f_i = A.r \). Note especially that entry retrieval is always determined by existing credential and ismem information, allowing selective credential retrieval:

\[
\text{credential } A Ra Re , \\
\text{entry } B Rb \text{ (role } A Ra \text{)} \rightarrow \\
\{ \neg \text{credential } B Rb \text{ (role } A Ra \text{)} \} .
\]

\[
\text{credential } A Ra Re , \\
\text{entry } B Rb \text{ (inter Res) ,} \\
\text{subject } Res \text{ (role } A Ra \text{)} \rightarrow \\
\{ \neg \text{credential } B Rb \text{ (inter Res)} \} .
\]

\[
\text{credential } A Ra Re , \\
\text{entry } B Rb (\neg A) \rightarrow \\
\{ \neg \text{credential } B Rb (\neg A) \} .
\]

\[
\text{credential } A Ra Re , \\
\text{ismem (role } D R \text{)} A , \\
\text{entry } B Rb \text{ (linked_role } D R Ra \text{)} \rightarrow \\
\{ \neg \text{credential } B Rb \text{ (linked_role } D R Ra \text{)} \} .
\]

Proving correctness of credential chain discovery is a matter of relating the algorithm with the specification [14; 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 formally isolate the entries 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 $\Sigma'$ be as given in Definition 3.2, let $\Gamma_{disc}$ contain auth and credential as defined above, and let $\Delta_{entry}$ contain linear entries. Suppose the query auth R A, top is successful, i.e. the judgement $J \triangleq \Sigma', \Gamma_{disc}; \Delta_{entry} \Rightarrow auth R A, top$ is derivable. Then the entries consumed by authorization is the sequence $\Delta$ such that $\Delta_{entry} = \Delta, \Delta'$ and $\Sigma', \Gamma_{disc}; \Delta \Rightarrow auth R A$ is a precedent in the last instance of the $\otimes_R$ rule in the derivation of $J$.

Soundness can then be demonstrated as follows. The result follows in a straightforward manner, since 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 $\Delta$ are the entries consumed by authorization, where:

$\Delta = \#linear entry A_1 R_1 R_{e1},\ldots,\#linear entry A_n R_n R_{en}$

Let:

$\Gamma_C = credential A_1 R_1 R_{e1},\ldots,credential A_n R_n R_{en}$

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

4.2.2 **Issuer-Driven Discovery.** Issuer-driven discovery works under the assumption that credentials $A.r \leftarrow f$ are stored with credential issuers. Hence, given an authorization query of the form auth $[A.r] B$, forward chain discovery starts by obtaining all entries that define the role $A.r$, which by convention are held by $A$. Subsequent credential discovery is then driven by rebuilding a chain of credentials in the issuer-to-subject direction. Hence, we define retrieve as follows:

retrieve (role A R) B A R Re <= entry A R Re.

and we define the following credential clauses. Note especially that entry retrieval is always determined by existing credential and ismem information, allowing selective credential 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}.

Proving correctness of backwards discovery is as straightforward as proving soundness of forwards discovery. As for the latter, this is because any credential atom is predicated by an existing entry assertion.
4.2.3 Bidirectional Discovery. In [14], it is observed that in realistic settings, credentials likely will sometimes be held with issuers, and sometimes with subjects. Thus, a bidirectional (mixed issuer- and subject-driven) strategy is likely practical. It is straightforward to obtain a mixed strategy in our setting by combining the clauses defined for the forward and backwards strategies. And again, soundness is trivial by composition. This technique differs from that in [14], in that the latter deterministically “switches” between retrieval of issuer- and subject-held credentials, by maintaining separate queues of subgoals. Our technique, on the other hand, is non-deterministic since subgoals will be chosen by the implementation. However, it would be straightforward to modify our mixed technique with two lists and an additional predicate that would enforce this alternation discipline. Other techniques to increase control over retrieval are discussed in Sect. 6.

5. RT FRAMEWORK VARIATIONS

In this section we implement variations on the basic RT system proposed in [13]. These implementations demonstrate the adequacy of LolliMon to express the RT framework generally, and also features of LolliMon that enhance previous specifications. Furthermore, the implementations provided here are the first implementations of chain discovery for variations of RT. However, because of the manner in which authorization is defined, i.e. as an entailment composition of credential retrieval and logical inference steps, it is essentially sufficient for us to extend the forward chaining specification in Sect. 3 with logical features of the variations. These extensions are modeled on similar extensions in to the Datalog specification of RT in [13].

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 [13], 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:

\[
\text{StateU}.\text{foundingAlumni} \leftarrow \text{StateU}.\text{diploma}(X, Y;[1955..1958])
\]

(1)

In RT_2, parameters may include entities optionally constrained to be in some role. Recalling example 4 from [13], we could state the policy that “Alpha company allows members of a project team to read documents of the project” as follows:

\[
\text{Alpha}.\text{file}(\text{read}, X:\text{Alpha}.\text{documents}(Y)) \leftarrow \text{Alpha}.\text{team}(Y)
\]

(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.

\[
p\text{constraint} : \text{type}.
\]

\[
\text{credential} : \text{entity} \rightarrow \text{role_name} \rightarrow \text{role_expr} \rightarrow \text{list pconstraint} \rightarrow \text{o}.
\]

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):

\[
\text{intc} : \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{pconstraint}.
\]

\[
\text{osetc} : \text{entity} \rightarrow \text{role_expr} \rightarrow \text{pconstraint}.
\]

\[
\text{thisc} : \text{entity} \rightarrow \text{pconstraint}.
\]

These constraints are endowed with a straightforward interpretation:

\[
\text{interp} \ \text{nil} \ D.
\]

\[
\text{interp} \ (\text{intc} \ X \ B \ T::\text{Ps}) \ D \Leftarrow
\]

\[
\text{leq} \ B \ X,
\]

\[
\text{leq} \ X \ T,
\]

\[
\text{interp} \ \text{Ps} \ D.
\]
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, the linked role case would be defined as follows:

\[
\text{credential } A \text{ RO } (\text{linked\_role } B \text{ R1 R2 }) \text{ Cs},
\text{ismem } (\text{role } B \text{ R1 }) \text{ D},
\text{ismem } (\text{role } D \text{ R2 }) \text{ E},
\text{interp } Cs \text{ E } \Rightarrow
\{ !\text{ismem } (\text{role } A \text{ R0 }) \text{ E} \}.
\]

And so on for the other forms. The implementation only requires redefinition of the `entry` datatype as for `credentials`, and a modification of `auth` and `discovery` clauses to support `pconstraints`. For example, in forward-chaining discovery:

\[
\text{credential } A \text{ Ra Re Cs1},
\text{entry } B \text{ Rb } (\text{role } A \text{ Ra }) \text{ Cs2 } \text{o}
\{ !\text{credential } B \text{ Rb } (\text{role } A \text{ Ra }) \text{ Cs2} \}.
\]

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:

\[ Ryan \xrightarrow{\text{Registrar as CompSci.Enroll}} \text{enroll\_req} \]  

(1)

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

\[
\text{delegation} : \text{entity} \to \text{entity} \to \text{role}\_expr \to \text{entity} \to \text{o}. 
\]

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:

\[
\text{for\_role} : \text{entity} \to \text{entity} \to \text{role}\_expr \to \text{o}. 
\]

\[
\text{delegation} B1 D (\text{role} A R) B2, \\
\text{for\_role} B1 D (\text{role} A R) => \\
\{!\text{for\_role} B2 D (\text{role} A R)\}. 
\]

\[
\text{credential} A R (^ B) => \\
\{!\text{for\_role} B B (\text{role} A R)\}. 
\]

\[
\text{credential} A R0 (\text{role} B R1), \\
\text{for\_role} D E (\text{role} B R1) => \\
\{!\text{for\_role} D E (\text{role} A R0)\}. 
\]

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

\[
\text{ismem} R A <= \text{for\_role} A A R. 
\]

Authorization is then formalized as the ability to activate a role membership. In RT\textsuperscript{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 role membership in the activation, which can be established via previously discussed techniques. To wit:

\[
\text{auth} : \text{entity} \to \text{entity} \to \text{role}\_expr \to \text{o}. 
\]

\[
\text{auth} A B R o- \text{for\_role} A B R, \text{top}. 
\]

\[
\text{auth} A B R o- \\
\text{retrieve} R B D Rd RE, \\
\text{(credential} D Rd RE => \{\text{auth} A B R\}). 
\]

where retrieve and credential are as defined in the subject-driven, issuer-driven, or bidirectional schemes defined above. 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 credential establishing the membership of Registrar in CompSci.Enroll is an 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:

\[
\text{delegation} registrar registrar (\text{role} compsci enroll) ryan, \\
\text{delegation} ryan registrar (\text{role} compsci enroll) enroll\_req => \\
\text{auth} enroll\_req registrar (\text{role} compsci enroll). 
\]

6. CONCLUSION

A summary of related work and the contributions and technical developments in this paper is given in Sect. 1. We now conclude with some remarks on future work.
An obvious direction for future work is to engineer a scheme for non-local credential retrieval on the foundations described here. A method for implementing credential retrieval within LolliMon would be required. A realistic architecture would also 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 would be a thorough efficiency analysis of the LolliMon implementation of chain discovery, which, though modeled on the techniques in [14], may not inherit the same complexity characteristics due to differences in proof inference. Another related topic of interest is the efficiency and flexibility of proof direction 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 the best; for example, note that even bidirectional discovery only allows chains to be reconstructed from the ends towards the middle, and fails if credentials are stored in a less regular fashion. Furthermore, factors such as expected wait times can dictate which credentials are preferable to retrieve, and factoring this information into proof direction could significantly enhance efficiency. This is the goal of the system RT⁵ [5], the specification and discovery techniques of which can easily be framed within LolliMon, in the same manner as the RT variants studied in Sect. 5. We intend to investigate flexible and efficient credential retrieval strategies as future work.

REFERENCES

A LolliMon Foundation for Distributed Trust Management

A LolliMon SUMMARY

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

SYNTAX:

\[ A ::= P | \top | A_1 \& A_2 | A_1 \rightarrow A_2 | A_1 \Rightarrow A_2 | \forall x: \tau. A \{ \{ S \} \} \]

Unrestricted Context \[ \Gamma ::= \cdot \mid \Gamma, A \]

Linear Context \[ \Delta ::= \cdot \mid \Delta, A \]

Pattern Context \[ \Psi ::= \cdot \mid S, \Psi \]

SEQUENT FORMS:

\[ \Gamma; \Delta \Rightarrow A \] Right inversion
\[ \Gamma; \Delta \Rightarrow A \Rightarrow P \] Left focusing
\[ \Gamma; \Delta \Rightarrow A \Rightarrow S \] Right focusing
\[ \Gamma; \Delta \Rightarrow \forall x: \tau. A \] Left inversion

RIGHT INVERSION RULES:

\[ \frac{\Gamma, A; \Delta; A \Rightarrow P}{\Gamma; \Delta \Rightarrow A \Rightarrow P} \text{ atm} \]
\[ \frac{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R}{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R} \]
\[ \frac{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R}{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R} \]
\[ \frac{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R}{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R} \]
\[ \frac{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R}{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R} \]
\[ \frac{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R}{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R} \]
\[ \frac{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R}{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R} \]
\[ \frac{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R}{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R} \]
\[ \frac{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R}{\Gamma; \Delta \Rightarrow A \Rightarrow B \Rightarrow R} \]

LEFT FOCUSING RULES:

\[ \frac{\Gamma; \Delta; \cdot \Rightarrow A \Rightarrow \top}{\Gamma; \Delta \Rightarrow \top \Rightarrow \top} \text{ atm} \]
\[ \frac{\Gamma; \Delta; \cdot \Rightarrow A \Rightarrow \top}{\Gamma; \Delta \Rightarrow \top \Rightarrow \top} \text{ atm} \]
\[ \frac{\Gamma; \Delta; \cdot \Rightarrow A \Rightarrow \top}{\Gamma; \Delta \Rightarrow \top \Rightarrow \top} \text{ atm} \]
\[ \frac{\Gamma; \Delta; \cdot \Rightarrow A \Rightarrow \top}{\Gamma; \Delta \Rightarrow \top \Rightarrow \top} \text{ atm} \]
\[ \frac{\Gamma; \Delta; \cdot \Rightarrow A \Rightarrow \top}{\Gamma; \Delta \Rightarrow \top \Rightarrow \top} \text{ atm} \]
\[ \frac{\Gamma; \Delta; \cdot \Rightarrow A \Rightarrow \top}{\Gamma; \Delta \Rightarrow \top \Rightarrow \top} \text{ atm} \]
\[ \frac{\Gamma; \Delta; \cdot \Rightarrow A \Rightarrow \top}{\Gamma; \Delta \Rightarrow \top \Rightarrow \top} \text{ atm} \]
\[ \frac{\Gamma; \Delta; \cdot \Rightarrow A \Rightarrow \top}{\Gamma; \Delta \Rightarrow \top \Rightarrow \top} \text{ atm} \]

FORWARD CHAINING RULES:

\[ \frac{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S}{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S}{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S}{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S}{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S}{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S}{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S}{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S}{\Gamma; \Delta; A \Rightarrow \alpha \Rightarrow S} \text{ atm} \]

MONADIC LEFT FOCUSING RULES:

\[ \frac{\Gamma; \Delta; S \Rightarrow S}{\Gamma; \Delta; S \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; B \Rightarrow S}{\Gamma; \Delta; B \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \& B \Rightarrow S}{\Gamma; \Delta; A \& B \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \& B \Rightarrow S}{\Gamma; \Delta; A \& B \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \& B \Rightarrow S}{\Gamma; \Delta; A \& B \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \& B \Rightarrow S}{\Gamma; \Delta; A \& B \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \& B \Rightarrow S}{\Gamma; \Delta; A \& B \Rightarrow S} \text{ atm} \]
\[ \frac{\Gamma; \Delta; A \& B \Rightarrow S}{\Gamma; \Delta; A \& B \Rightarrow S} \text{ atm} \]
**Left Inversion Rules:**

\[
\begin{align*}
\Gamma; \Delta; A; \Psi & \rightarrow S \\
\Gamma; \Delta; A; \Psi & \rightarrow S \quad \text{async} \\
\Gamma; \Delta; 1; \Psi & \rightarrow S \\
\Gamma; \Delta; S_1; S_2; \Psi & \rightarrow S \quad \text{\( \otimes \)} \\
\Gamma; \Delta; \exists_x \tau; S'; \Psi & \rightarrow S \\
\Gamma; \Delta; ! A; \Psi & \rightarrow S \quad L
\end{align*}
\]

**Right Focusing Rules:**

\[
\begin{align*}
\Gamma; \Delta & \Rightarrow A \\
\Gamma; \Delta & \Rightarrow A \\
\Gamma; \Delta & \Leftrightarrow 1
\end{align*}
\]

\[
\begin{align*}
\Gamma; \Delta_1 & \Rightarrow S_1 \\
\Gamma; \Delta_2 & \Rightarrow S_2 \\
\Gamma; \Delta & \Rightarrow \exists x \tau; S
\end{align*}
\]

**B. Example Derivation**

To illustrate the LolliMon derivation system, we show the proof of \( a \vdash \{ b \}; a, a \Rightarrow \{ a \otimes b \} \) which we may intuitively think of as asking whether the unrestricted formula \( a \vdash \{ b \} \) (intuitively thought of as a rewrite rule turning \( a \) into \( b \)) can transform the linear context \( a, a \) (representing a state with just \( a \) and \( b \)). Although this judgement is derivable by applying \( a \vdash \{ b \} \) once, the LolliMon implementation will not be able to find the derivation. The eager forward chaining semantics will cause \( a \vdash \{ b \} \) to be applied twice before reverting to backchaining, thus preventing the goal \( a \otimes b \) from being solved, demonstrating incompleteness of the implementation. The first derivation elaborates the elision in the second.

\[
\begin{align*}
\frac{a \vdash \{ b \}; a \Rightarrow a \quad (\text{uhyp, atm})}{a \vdash \{ b \}; a \Rightarrow a \otimes b \quad (\text{\( \otimes \)}}} \\
\frac{a \vdash \{ b \}; a, b \Rightarrow a \otimes b \quad (\text{uhyp, atm})}{a \vdash \{ b \}; a, b \Rightarrow a \otimes b \quad \Rightarrow \rightarrow \quad \{ \} \quad \text{L, async, \( \rightarrow \rightarrow \)}
\end{align*}
\]

\[
\begin{align*}
\frac{a \vdash \{ b \}; a, b; \Rightarrow a \otimes b \quad (\text{uhyp, atm})}{a \vdash \{ b \}; a, a \Rightarrow a \otimes b \quad \Rightarrow \rightarrow \quad \{ \} \quad \text{R}}
\end{align*}
\]

**C. 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 C.1.**

(1) If \( \Gamma, P; \vdash S \) then \( \Gamma; \vdash \{ ! P \} > S \).

IEEE Symposium on Security and Privacy
\(2\) \(\Gamma, P; \Rightarrow P\)

\(3\) \(\Gamma, P; \rightarrow P\)

**Proof of Lemma 3.1.**

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

**Case:** \(\Sigma', \Gamma_C, \Gamma; :: \text{ismem' role} > \text{ismem (role A R)} B\)

\(\Sigma', \Gamma_C, \Gamma; :: \text{cred A R (role A' R')} B\) and \(\Sigma', \Gamma_C, \Gamma; :: \text{ismem (role A' R')} B\) by inversion,

\(\Sigma, \Gamma_C; :: \text{cred A R (role A' R')} B\) by assumption,

\(\Sigma, \Gamma_C; :: \text{ismem (role A' R')} B\) by inversion (note clauses in \(\Sigma'\) are all monadic and not applicable),

\(\Sigma, \Gamma_C; :: \text{ismem (role A' R')} B\) by assumption,

\(\Sigma, \Gamma_C; :: \text{ismem Res B}\) by induction hypothesis

where

\(\text{ismem' role} = \text{ismem (role A' R')} B \odot \text{cred A R (role A' R')} \odot \text{ismem (role A R)} B\)

\(\text{ismem role} = \text{ismem (role A' R')} B \circ \text{cred A R (role A' R')} \circ \text{ismem (role A R)} B\)

Part 2. By structural induction on the given derivation making use of the assumption on \(\Gamma\).

**Case:** \(\Sigma', \Gamma_C, \Gamma; :: \text{ismems cons} \gg \text{ismems (role A R::Res)} B\)

\(\Sigma', \Gamma_C, \Gamma; :: \text{ismem (role A R)} B\) and \(\Sigma', \Gamma_C, \Gamma; :: \text{ismems Res B}\) by inversion,

\(\Sigma, \Gamma_C; :: \text{ismem Res B}\) by induction hypothesis

\(\Sigma, \Gamma_C; :: \text{ismems cons} \gg \text{ismems (role A R::Res)} B\) by \(\supset\),

\(\Sigma, \Gamma_C; :: \text{ismems (role A R::Res)} B\) by uhyp

where

\(\text{ismems cons} = \text{ismem (role A R)} B \odot \text{ismems Res B} \odot \text{ismems (role A R::Res)} B\)

\(\text{ismem' cons} = \text{ismem (role A R)} B \circ \text{ismems Res B} \circ \text{ismems (role A R::Res)} B\)

\(\Box\)

**Proof of Theorem 3.1.** By inversion on the given derivation and an appeal to lemma 3.1. \(\Box\)

**Proof of Lemma 3.2.** By structural induction on the first given derivation. There are 4 cases to consider, one for each clause in \(\Sigma\).

**Case:** \(\Sigma, \Gamma_C; :: \text{ismem role} \gg \text{ismem (role A R)} B\) and \(\Sigma', \Gamma_C, \Gamma; :: \text{ismem (role A R)} B; :: \rightarrow S\)

then

\(\Sigma, \Gamma_C; :: \text{cred A R (role A' R')} B\) and \(\Sigma, \Gamma_C; :: \text{ismem (role A' R')} B\) by inversion,

\(\Sigma, \Gamma_C, \Gamma; :: \text{cred A R (role A' R')} B\) by lemma C.1,

\(\Sigma', \Gamma_C, \Gamma; :: \text{ismem (role A' R')} B; :: \text{ismem role} \gg S\) by lemma C.1,

\(\Sigma', \Gamma_C; :: \text{ismem (role A' R')} B; :: \text{ismem' role} \gg S\) by weakening,

\(\Sigma', \Gamma_C; :: \text{ismem (role A' R')} B; :: \text{ismem' role} \gg S\) by \(\supset\),

\(\Sigma', \Gamma_C; :: \text{ismem (role A' R')} B; :: \rightarrow S\) by uhyp,
\begin{align*}
\Sigma', \Gamma_C, \Gamma; \vdash S & \text{ by induction hypothesis} \\
\text{where} & \\
\text{ismem\_role} &= \text{ismem (role A' R')} B \circ \text{cred A R (role A' R')} \circ \\
\text{ismem (role A R) B} & \\
\text{ismem\'_role} &= \text{ismem (role A' R')} B \circ \text{cred A R (role A' R')} \circ \\
\{ \text{ismem (role A R) B} \}
\end{align*}

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. \hfill \square

**Proof of Theorem 3.2.** Direct from lemma C.1 and lemma 3.2 as follows:

\begin{align*}
\Sigma, \Gamma_C; & \Rightarrow \text{ismem (role A R) B} \text{ by assumption.} \\
\Sigma', \Gamma_C, \text{ismem (role A R) B}; & \Rightarrow \text{ismem (role A R) B} \text{ by lemma C.1.} \\
\Sigma', \Gamma_C; & \Rightarrow \text{ismem (role A R) B} \text{ by lemma 3.2.} \\
\Sigma', \Gamma_C; & \Rightarrow \{ \text{ismem (role A R) B} \} \text{ by } \{ \}^R. \\
\end{align*} \hfill \square