DKAL: Distributed-Knowledge Authorization Language

Yuri Gurevich
Microsoft Research
One Microsoft Way
Redmond, WA 98052
gurevich@microsoft.com

Itay Neeman[*]
Department of Mathematics
University of California Los Angeles
Los Angeles, CA 90095-1555
ineeman@math.ucla.edu

Abstract

DKAL is a new declarative authorization language for distributed systems. It is based on existential fixed-point logic and is considerably more expressive than existing authorization languages in the literature. Yet its query algorithm is within the same bounds of computational complexity as e.g. that of SecPAL. DKAL's communication is targeted which is beneficial for security and for liability protection. DKAL enables flexible use of functions; in particular principals can quote (to other principals) whatever has been said to them. DKAL strengthens the trust delegation mechanism of SecPAL. A novel information order contributes to succinctness. DKAL introduces a semantic safety condition that guarantees the termination of the query algorithm.

1. Introduction

In an increasingly interconnected world, the authorization policies grow more involved. Rights assigned and maintained by an autonomous central authority give way to rights that depend upon credentials issued by outside entities that may rely upon credentials issued by yet other entities. An authorization language should handle all that in a secure, uniform and comprehensible way amenable to analysis. It should facilitate policies that are more modular and thus more stable in the changing environment.

Logic is a natural foundation for declarative authorization languages. It allows one to write high-level policy rules in a human-readable form. The resulting declarative policy serves as a base, a legal manifesto of sorts, from which specific permissions are derived. And indeed many logic-based authorization languages have been proposed. One of the latest is SecPAL [5]; a quick review of preceding languages is found in [5, §8]. Here we introduce Distributed Knowledge Authorization Language, in short DKAL, conceived in Fall 2006 when SecPAL appeared [4] and Itay Neeman first visited Microsoft Research. Is there a need in another authorization language? We believe so. Here are some of the reasons.

1. There is a potential information leak problem in SecPAL and all preceding languages. A naive dramatization in Fig. 1 illustrates the problem in SecPAL terms. The department of Special Operations of some intelligence agency appoints secret agents by assertions like S1. Bob, who is just a receptionist, wants to find out who secret agents are. He does not dare to pose that query (and suspects that the system would not allow him to); instead he asserts S2 and S3 where spot 97 is one of the parking spots over which he has the authority, e.g. a visitor spot. It follows from S1 and S2 that `Bob says John Doe isSecrAgent`. Now, by posing an "innocent" query about who can park in spot 97, Bob gets a list of all secret agents. The problem can be addressed on the level of implementation, for example by attempting to separate confidential and non-confidential information (which is easier said than done; both may be necessary to derive certain permissions), but the right way to address the problem is at the authorization-language level. DKAL solves the problem by making communication targeted. The analog of the naive dramatization does not work in DKAL as assertions like S1 would be targeted to an audience that excludes Bob. See more on info leak in §4.

2. The expressivity of the existing languages is too limited. Consider for example nested quotations. They are expressible in Speaks-For [1], which has expressivity limitations of its own, but not in SecPAL and other Datalog based languages. More generally, following Datalog, these languages do not use functions in arguments of recursively defined relations. Avoiding functions (if and when it is possible) can make policies more awkward and less natural. In principle, Datalog with constraints can simulate such use of functions (compare the two programs of §2) but this would violate the feasibility restrictions of the languages in ques-

[*]Most of the work presented here was done when Neeman was a Visiting Researcher at Microsoft Research.

978-0-7695-3182-3/08 $25.00 © 2008 IEEE
DOI 10.1109/CSF.2008.8

```
SpecialOperations says John Doe isSecrAgent                    (S1)
Bob says SpecialOperations can say p isSecrAgent               (S2)
Bob says p canParkInSpot 97 if p isSecrAgent                   (S3)
```

Figure 1. Secret agent information leakage

tion. DKAL enables unrestricted use of functions that can be nested and mixed while maintaining the computational time bounds of SecPAL. (The decidability and complexity issues are addressed in §9 and §10.)

3. One can make authorization rules more succinct by partially ordering portions of information independently of who possesses them. §7 is devoted to the information order which is denoted $\leq$; $x \leq y$ implies that y is at least as informative as x. The information order is defined recursively. This recursion is powerful, which is especially useful in the context of nested expressions. See for example rule SaidMon in Fig. 5 which is a part of the definition of the information order on quotations.

4. There is a better logical platform for authorization languages than Datalog with or without constraints, namely existential fixed-point logic (EFPL). We recall EFPL in §2. We think also that knowledge is so important in authorization theory that it should be made explicit.

DKAL addresses all these concerns. It is more expressive than the languages in the literature. In §11 we give a natural embedding of SecPAL, one of the most expressive authorization languages to date, into DKAL. §9 and §10 are devoted to a query evaluation algorithm for DKAL, with the same time bounds as SecPAL's query evaluation algorithm. Before reaching those sections, we illustrate DKAL by examples, define it precisely, and discuss various aspects including targeted communication, the use of functions, the information order, and DKAL's mechanism for ensuring termination of the query evaluation algorithm in the presence of functions. We discuss related work in §12, and we conclude in §13 with a summary and directions for future work.

Acknowledgements

We gratefully acknowledge collaboration with M. Paramasivam on earlier stages of the project, conversations with Slava Kavsan, and comments of Moritz Becker, Nikolaj Bjørner and Andreas Blass.

2. Existential fixed-point logic (EFPL)

We recall what EFPL is and introduce the substrate/superstrate terminology. EFPL was introduced in [8] and has an attractive model theory. A version of EFPL in the form related to Constrained Logic Programming [17] is given in [15, Appendix A]. That version is easy to introduce to an audience familiar with logic programming in the form of Datalog with Constraints [20] or Pure Prolog [26]. In either case, a logic program is used to define additional relations over a given first-order structure. In our terminology, the given structure is the *substrate structure* or just the *substrate*, and the new relations are *superstrate relations*. In the Constraint Datalog case, the substrate is often called the constraint domain [20]. In the Pure Prolog case, the substrate includes a Herbrand universe with no built-in relations, with the possible exception of equality; in addition, the substrate may have a limited number of additional datatypes, e.g. integer arithmetic.

EFPL employs logic programs that can be defined as generalized Constraint Datalog programs. First, we need to remove the restriction on the use of function symbols. A Datalog rule has the form:

$$R_0(s_1, \ldots, s_j) \mathrel{:-} R_1(t_1, \ldots, t_k), R_2(u_1, \ldots, u_\ell), \ldots, \mathit{con}$$

where the arguments of relation symbols R_i are variables or constants and where *con* is a quantifier-free first-order formula in the substrate vocabulary. EFPL allows the arguments to be arbitrary expressions (a.k.a. terms) in the substrate vocabulary. Second, in the literature, there are always limitations on the legal substrates of Constraint Datalog which are imposed to guarantee good algorithmic properties of the legal substrates. For example, [20] requires quantifier elimination. EFPL imposes no such limitations.

Alternatively, EFPL logic programs can be defined as generalizations of Pure Prolog programs where a substrate is an arbitrary first-order structure. In particular, the substrate may have free constructors (like the functions of a Herbrand universe) applied to regular elements, that is, elements that are not produced by means of free constructors.

In the sequel, logic programs are by default EFPL programs. A logic program Π over a substrate X computes the superstrate relations and thus produces an enriched structure $\Pi(X)$. This is clear if Π terminates over X. If Π does not terminate over X, it still computes the superstrate relations. It just takes infinite time to do so. Fortunately, every particular instance of a superstrate relation is computed at a finite stage. In cases of interest in the rest of this paper, the programs compute all relevant instances in finite time,

```
Chux:    (Alice canDownload Article) to Alice                                  (A1)
Alice:    Best tdOn Alice canDownload Article                                  (A2)
Best:    (Chux tdOn p canDownload Article) to p                                (A3)
a knows x ← a knows p said x, a knows p tdOn x                                 (C1)
a knows p tdOn (q tdOn x) ← a knows p tdOn x, a knows q exists                 (C2)
a knows p tdOn (q tdOn0 x) ← a knows p tdOn x, a knows q exists
Alice knows Chux said Alice canDownload Article                                (K1)
Alice knows Best tdOn Alice canDownload Article                                (K2)
Alice knows Best said (Chux tdOn Alice canDownload Article)                    (K3)
Alice knows Chux exists                                                        (K4)
Alice knows Best tdOn (Chux tdOn Alice canDownload Article)                    (K5)
Alice knows Chux tdOn Alice canDownload Article                                (K6)
Alice knows Alice canDownload Article                                          (K7)
```

Figure 2. User centric delegation example

so we need not worry about non-terminating computations.

For example, consider this simple logic program:

$$T(\texttt{left}(x), \texttt{right}(x))$$
$$T(\texttt{right}(x), \texttt{left}(y)) \leftarrow T(x, y)$$
$$T(x, z) \leftarrow (T(x, y) \wedge T(y, z))$$

where `left` and `right` are substrate functions, and T is a superstrate relation. If we assume that the two functions are free constructors and that there is at least one substrate constant, so that the corresponding Herbrand universe is well defined, then our logic program is a Pure Prolog program. In that case, the substrate is the infinite binary tree, and the program computes a partial order T that is the lexicographical order at every level of the tree. But the program is meaningful without these additional assumptions.

As written, the example program is not a Constraint Datalog program because the function symbols occur in rule heads. Define Liberal Datalog as a Constraint Datalog with no limitation on the legal substrates. The example program reduces to the following Liberal Datalog program:

$$T(u, w) \leftarrow u = \texttt{left}(x) \wedge w = \texttt{right}(x)$$
$$T(u, w) \leftarrow T(x, y) \wedge u = \texttt{right}(x) \wedge w = \texttt{left}(y)$$
$$T(x, z) \leftarrow T(x, y) \wedge T(y, z)$$

Furthermore, in a similar way, any EFPL program can be transformed into a Liberal Datalog program.

In addition to logic programs, EFPL has queries. Queries are first-order formulas, subject to some restrictions, in the substrate vocabulary enriched with superstrate symbols. Queries are evaluated not in a given substrate X, but in the structure $\Pi(X)$ produced by the given logic program Π over the substrate X. The precise syntax of EFPL queries is immaterial here. DKAL queries are defined in §8.4.

As we mentioned already, there are different forms of Constraint Datalog in the literature, distinguished by the safety restrictions they place on programs to ensure termination. The form obtained from EFPL with the DKAL safety conditions is new. We'll say more on how DKAL guarantees termination in §9.

3. A user centric example

DKAL's syntax (vocabulary, rules, assertion forms) is defined in §8. Here we begin introducing it gradually, by examples. Since SecPAL is naturally translated into DKAL, all the varied scenarios of [4, Section 5] are expressible in DKAL. So we give examples of different kinds in this paper. We start with a user-centric example, partially because we believe that DKAL is particularly appropriate for the user centric approach to authorization. We demonstrate the basics of DKAL, in particular how trust and delegation are expressed.

Alice would like to download Article from Repository in course of her work for Fabricam. Repository lets Fabricam employees download content with no constraints. Fabricam in turn requires that its employees respect intellectual property. Fig. 2 shows how Alice verified her right to download Article.

Alice bought the right at an online store Chux (an allusion to Chuck's). Chux told her that she can download Article; this is represented by assertion A1 in Fig. 2. In the formal model we compute a superstrate relation `knows`, and the assertion A1 leads

```
Chux:  (a canDownload s) to a ←                                          (A4)
   a authorized $k to Chux for s, a hasPayRate Perfect, price(s) = k.
accounts.Chux:    (Alice hasPayRate Perfect) to Chux                     (A5)
Chux:  accounts.Chux tdOn a hasPayRate e                                 (A6)
Chux:  a tdOn (a authorized $k) to Chux for s                            (A7)
Alice:  (Alice authorized $40 to Chux for Article) to Chux               (A8)
```

Figure 3. Confidentiality example

to the instance K1 of that relation. The expression `Alice canDownload Article` denotes an *infon*, a piece of information, and so does `Chux said Alice canDownload Article`. The relation `knows` is of type Principal × Info. Note that from assertion A1 Alice learns only that `Chux said Alice canDownload Article`, not that `Alice canDownload Article`.

Alice noticed that the copyright for Article belongs to Best Publishing House; hence the assertion A2 where `tdOn` stands for *is trusted on*. The expression `Best tdOn Alice canDownload Article` denotes yet another infon, and assertion A2 leads to instance K2 of `knows` in the formal model.

The intended meaning of p `tdOn` x is given by two rules. One is C1 which states that a principal a knows x if she knows that some principal p said x and that p is trusted on x. We'll get to the other rule shortly.

Unfortunately Alice does not know whether Chux can be trusted on `Alice canDownload Article`, and Best, who is trusted, did not say that `Alice canDownload Article`. So Alice cannot yet conclude that she is allowed to download Article. Alice contacts Best who authorized Chux to sell download rights to Article and who has in its policy the assertion A3 (with a free, unconstrained variable p). As a result Alice learns K3.

The infon p `tdOn` x expresses not only trust in p on x, but also a permission for p to delegate the trust. (There is a way to express non-delegatable trust, using $\texttt{tdOn}_0$ instead of `tdOn`. The distinction between `tdOn` and $\texttt{tdOn}_0$ is inherited from SecPAL and will be addressed later.) The right to delegate is captured by the double rule C2; only the first line is relevant to the current example. If a knows that p `tdOn` x and that q `exists` then a knows that p is also trusted on q `tdOn` x, and this allows p to delegate the trust to q. The restriction that a knows (the existence) of q is a safety condition that prevents the knowledge of a from exploding with irrelevant details. We'll say more about the rule that leads to knowledge of infons of the form q `exists` in §9; here it suffices to say that the rule applies to K1 and results in K4.

Applying rules C1 and C2 to K1–K4, Alice obtains K5–K7. Having deduced K7, Alice approaches Repository, and downloads Article.

4. Info leak, and targeted communication

Recall the naive dramatization of the information leakage problem in §1. Let's consider a slightly less naive example. Modify the scenario of §3 by replacing assertion A1 with the assertions in Fig. 3. Chux compiles payment statistics of customers and rates them. Customers rated "perfect" get the download authorization immediately upon authorizing a proper payment to Chux, even before the funds are received. The rating is managed by accounts.Chux. It is intended that customers know nothing about the rating system or their ratings or other customers' ratings. Chux makes assertion A4 with three conditions. A condition is an expression of type Info or a substrate constraint. In this case the first two conditions are infon expressions, and the third is a constraint using a substrate function `price`. Implicitly the assertion has also safety conditions, addressed in §8.3 and §9, that restrict the ranges of variables; the safety constraints apply also to assertions A6 and A7. According to assertion A5, accounts.Chux rated Alice "perfect"; note that the assertion is targeted only to Chux. According to assertions A6 and A7, Chux trusts accounts.Chux on payment ratings and trusts the customers on payment authorization. The price of Article is $40. When Alice decides to purchase Article, she makes the assertion A8. Assertions A4–A8 lead Chux to communicate the infon `Alice canDownload Article` to Alice; as above Alice can proceed to verify her right to download Article.

No infon of the form p `hasPayRate R` is communicated to Alice, and a probing attack such as the one in Fig. 1 does not work. The DKAL parallel of S2 here is assertion `Alice: accounts.Chux tdOn` p `hasPayRate Perfect`. The assertion is harmless, since A5 makes the infon `accounts.Chux said Alice hasPayRate Perfect` known only to Chux, not to Alice. The confidentiality of pay ratings of other principals is similarly protected.

The targeting of communication is beneficial also with respect to liability. Suppose that an agency A of state S_1 issues David a document, addressed to S_1 wine shops, that allows them to sell wine to David. If David buys alcohol from a wine shop in state S_2 and if this violates the law of S_2, agency A is not liable because it addressed the documents to wine shops in S_1, not in S_2.

Audience restrictions can be communicated by means of SAML [27], see specifically [11, §2.5.1.4]. (The issue is addressed in the SecPAL implementation as well.) While the audience restriction may be helpful with respect to liability, the SAML audience field does not solve the problem in Fig. 1. Indeed, if the fact `John Doe isSecrAgent` is modified with an audience restriction then all that Bob has to do is to use the modified fact in S2.

In probing attacks of the kind illustrated in Fig. 1, principals that are allowed to authorize some permissions leverage the authority to learn information they are not meant to know. One way to thwart such probing attacks is to disallow conditional assertions by "outsiders" (like Bob), as in Cassandra [6, 7], but this is too restrictive. One may filter out some conditional assertions on a case by case basis at the implementation level, but this ad-hoc approach makes it hard to reason about security. Yet another way is to compartmentalize facts to the extent possible and handle requests using primarily the relevant compartment policy. But there are limits to compartmentalization (unless you really have a union of essentially disjoint policies), principals still can probe facts in their compartments, and the approach does not make reasoning about security easy.

By targeting communication and separating knowing from saying, DKAL solves the problem at the level of the authorization language, so that information does not have to be compartmentalized a priori, and conditional assertions do not have to be filtered out. Of course DKAL does not prevent information from leaking as a result of negligence. For example, in the pay rate scenario, Chux may accidentally target Bertha's pay rating to Alice. But that is a very different story.

5. Use of functions

In Datalog, with or without constraints, the symbols of recursively defined relations are applied only to (tuples of) variables and individual constants, and the (Constraint) Datalog based authorization languages inherit the restriction on the use of functions. In contrast, existential fixed-point logic allows free use of function symbols, and EFPL based DKAL makes intensive use of functions, both user-specific and built-in. Function symbols routinely appear in the heads of rules, and typically our functions are free constructors. The flexible use of functions comes for a price. The proof of program termination, let alone complexity proofs, becomes much harder.

The built-in free-constructor functions include `said` and `tdOn`. Function `said` enables (possibly nested) quotations in authorization policies, which leads to greater flexibility in designing more modular policies. Suppose for example that Chux (which appeared in Figures 1 and 2) has several discount plans, and that employees of Fabricam participate in discount plan 5X4302. To obtain the discount, they must present a signed certificate from Fabricam stating that they are employees. Chux relies on a cryptographic server Crypto to verify that the signed statements are authentic. The system should be designed so that Crypto just verifies authenticity. Crypto's actions should not depend on Chux's policy on discounts, so that Chux's policy could be changed without requiring a change in Crypto's behavior.

Chux makes a quotation assertion A9 in Fig. 4. Crypto acts as a "dumb" server, merely decrypting the statements it receives, and passing them on to Chux. Policy, for example assertions A10 and A11, is the prerogative of Chux. In this case, the end effect (of authorizing the discount to Fabricam employees) could be achieved without quotations. Chux could trust Crypto on q `is an employee of Fabricam`, and Crypto in its own policy could trust Fabricam on this. The issue here is not just achieving the end effect, but the flexibility to concentrate the policy at one place.

DKAL's vocabulary may be extended by user-introduced functions and relations. We already saw function `price` in §4. Other typical user-introduced functions and relations relate to time, various directory structures, basic arithmetical operations, etc. DKAL also permits user-introduced functions that take attribute or infon values. To demonstrate this, modify the confidentiality example above by replacing assertions A7 and A8 with assertions A12–A14 in Fig. 4. Chux does not simply accept infon a `authorized $`k `to Chux for` s from customer a, but requires that the infon comes with a certificate, signed using a's private key. (Chux will need the certificate to obtain the funds from a bank.)

We assume here a given (that is substrate) relation `authentic`(a, x, c) meaning that c is a certificate of infon x signed with the private key of a. Given an infon x and string c, function `augm`(x, c) (an allusion to "augment") produces a new infon. When Alice wishes to purchase Article, she makes assertion A14, where C is a certificate of the infon `Alice authorized $40 to Chux for Article`, which Alice produced and signed using her private key. Then, due to assertions A12 and A13, `Chux knows Alice authorized $40 to Chux for Article`, and then, as in §4, Alice receives an authorization to download Article.

Chux: Crypto tdOn r said q is an employee of r (A9)

Chux: Fabricam tdOn q is an employee of Fabricam (A10)

Chux: q can take discount 5X4302 $\leftarrow$ q is an employee of Fabricam (A11)

Chux: a tdOn augm(a authorized \$$k$ to Chux for s, c) (A12)

Chux: a authorized \$$k$ to Chux for s $\leftarrow$ (A13)
augm(a authorized \$$k$ to Chux for s, c),
authentic(a, a authorized \$$k$ to Chux for s, c)

Alice: augm(Alice authorized \$40 to Chux for Article, C) to Chux (A14)

Chux: Crypto tdOn$_0$ r said q is an employee of r (A15)

Crypto:$_0$ (Fabricam said Chris is an employee of Fabricam) to Chux (A16)

a knows x $\leftarrow$ a knows p said$_0$ x, a knows p tdOn$_0$ x (C3)

Chux knows Crypto said$_0$ Fabricam said Chris is an employee of Fabricam (K8)

Chux knows Fabricam said Chris is an employee of Fabricam (K9)

Crypto:$_0$ (r said q is an employee of r) to Chux $\leftarrow$ (A17)
Crypto2 said r said q is an employee of r

Figure 4. Use of functions, and restricted delegation

6. Restricted delegation

One of the major advances of SecPAL [4] is the mechanism of restricted delegation. We adapted that mechanism to DKAL. DKAL has two kinds of infons expressing trust, p tdOn x, and p tdOn$_0$ x. The trust given by the former is delegatable; the trust given by the latter is not. To illustrate the use of non-delegatable trust, replace assertion A9 in Fig. 4 with assertion A15 in Fig. 4. The new assertion expresses non-delegatable trust in Crypto on r said q is an employee of r. Suppose that Crypto is given a signed certificate from Fabricam attesting that Chris is a Fabricam employee. After authenticating the certificate, Crypto produces assertion A16. The subscript 0 in A16 signifies restricted communication; more on this in the next paragraph. Assertion A16 leads to knowledge K8, with the subscript 0 on the first said. K8 and assertion A15 give K9 by means of rule C3.

The delegation rule C2 has delegatable trust assumed in its body and cannot be applied to A16, so Crypto cannot directly delegate the trust to others. He may attempt to circumvent the prohibition, for example by placing assertion A17. It seems that by saying the appropriate thing, Crypto2 enables A16. But the attempt fails because assertion A17 is restricted. The precise meaning of restricted assertion involves relation knows$_0$, read knows internally. p knows x internally if this follows from assertions placed by p himself, with no dependence on assertions placed by other principals. Restricted assertions can be conditioned only upon internal knowledge (see the final paragraph on §8.3) while A17 is based on communication from Crypto2 to Crypto.

We sometimes write knows$_\infty$, said$_\infty$, and tdOn$_\infty$ for knows, said, and tdOn. The distinction between knows$_\infty$ and said$_\infty$ on one side and knows$_0$ and said$_0$ on the other side is similar to SecPAL distinction between $\mathrm{AC}, \infty \models A$ says x and $\mathrm{AC}, 0 \models A$ says x, and is used here to the same effect, namely preventing principals from circumventing non-delegatability. Delegations of arbitrary bounded depth can be obtained by nesting tdOn$_0$ in the head of the assertion delegating the right. SecPAL examples on bounded depth delegation, see e.g. [5, §5] become DKAL examples via the embedding of SecPAL into DKAL explained in §11.

7. Information order

Rules C1–C3 have a common aspect: a principal a knows some infon x because a knows some other infons $y_1, \ldots, y_k$. The information order x ensues y (symbolically $x \le y$) on infons extracts the common aspect. (We resurrect the obsolete transitive meaning of ensue [28].) Ideally, the meaning of $x \le y$ would be that all information of x is present in y but this leads to undecidability. The actual order is a constructive approximation of the ideal one. The mediating rules KMon and KSum in Fig. 5 express the common aspect of C1–C3 and their counterparts for knows$_0$. Rule KMon states that knowledge of x is a

$$
\begin{array}{rcll}
a\ \texttt{knows}_d\ x & \leftarrow & a\ \texttt{knows}_d\ y,\ x \le y & \text{(KMon)}\\
a\ \texttt{knows}_d\ x_1 + x_2 & \leftarrow & a\ \texttt{knows}_d\ x_1, a\ \texttt{knows}_d\ x_2 & \text{(KSum)}\\
x \le p\ \texttt{said}_d\ x\ \texttt{+}\ p\ \texttt{tdOn}_d\ x & & & \text{(TrustApp)}\\
p\ \texttt{tdOn}\ (q\ \texttt{tdOn}_d\ x) \le p\ \texttt{tdOn}\ x\ \texttt{+}\ q\ \texttt{exists} & & & \text{(Del)}\\
p\ \texttt{tdOn}_0\ x \le p\ \texttt{tdOn}\ x & & & \text{(Trust0}\infty\text{)}\\
p\ \texttt{said}_d\ x \le p\ \texttt{said}_d\ y & \leftarrow & x \le y & \text{(SaidMon)}
\end{array}
$$

Figure 5. House Rules, part I

consequence of knowledge of y if x ensues y. Rule KSum introduces infon addition operation of type Info$\times$Info $\rightarrow$ Info, and the rule states that knowledge of $x_1 + x_2$ is a consequence of knowledge of both x_1 and x_2. Each of KMon and KSum is a *double rule*, with $d \in \{0, \infty\}$. We use double rule notation similarly below.

The content of rules C1 and C3 is now expressed succinctly by ensue double rule TrustApp. Similarly the content of rule C2 is expressed by ensue double rule Del. Rules KMon–Del are *house rules* of DKAL. Rules C1–C3 are not house rules; they are consequences of house rules.

The inclusion of the information order allows creating a rich structure of information with easily understood rules. For example rule Trust0∞ expresses the fact that non-delegatable trust is a consequence of delegatable trust. The inclusion of the information order also allows for easily expressing strong quotation semantics. The deceptively simple rule SaidMon incorporates consequences of speeches into the calculation of knowledge, so that, for example p `said` q $\texttt{tdOn}_0$ x ensues p `said` q `tdOn` x. DKAL thus has very strong semantics for quotations, computing not only principals' speeches, but also their implied consequences. The rule could not be expressed as a single rule without the information order .

8. The nuts and bolts

8.1. Substrate

Substrate and superstrate were mentioned already §2. In DKAL, a substrate is a many-sorted structure X satisfying certain requirements that we describe in this section. The basic functions and relations of X are *substrate functions* and *substrate relations*. The structure X can be partial in the sense that substrate functions can be partial. The possible partiality results in some details that one has to be cautious about. In this exposition, for simplicity, we ignore those details; none of our results is compromised by that. The vocabulary of X, the *substrate vocabulary*, does not contain any of the five superstrate relation symbols described in the next subsection.

We assume that substrate functions and relations are computable. More precisely, we assume that substrate elements are (represented by) strings in a fixed alphabet and that there is an algorithm Eval that evaluates substrate functions and relations. Given a function name F of arity h and elements $a_1, \ldots, a_j$, Eval computes $F(a_1, \ldots, a_j)$. We treat constants as nullary functions. Given a relation name R of arity j and elements $a_1, \ldots, a_j$, Eval determines whether $R(a_1, \ldots, a_j)$ is true or false.

The universe of X splits into two sorts. One is Regular, with a subsort Principal and possibly other, user defined, subsorts. Regular elements may be principals, time moments, time intervals, files, directories, domain names, etc. The other is Synthetic, with subsorts Attribute, Speech, and Info. Functions with regular (resp. synthetic) values are regular (resp. synthetic), and the same convention applies to variables and expressions in general. (Here and below, expressions are by default first-order expressions, that is, first-order terms, in the substrate vocabulary.) Every synthetic function is a free constructor, and every synthetic element is constructed, in a unique way, from regular elements by means of synthetic functions. The *semantic tree* of a substrate element b is the unique ordered finite tree rooted at b and such that

- if b is regular then semtree(b) has no other nodes,
- if $b = F(b_1, \ldots, b_n)$ and function F is synthetic then there are exactly n subtrees under the root: semtree$(b_1), \ldots,$ semtree(b_n).

A substrate relation a `regcomp` b holds if and only if a is regular, b is synthetic, and a is a leaf of semtree(b). A *syntactic tree* of an expression t is the unique ordered finite tree rooted at t and such that

- if t is regular, then syntree(t) has no other nodes,
- if $t = F(t_1, \ldots, t_n)$ and function symbol F is synthetic, then there are exactly n subtrees under the root: syntree$(t_1), \ldots,$ syntree(t_n).

Regular Sort	::=	Regular $\mid$ Principal $\mid$ $\ldots$
Synthetic Sort	::=	Synthetic $\mid$ Info $\mid$ Speech $\mid$ Attribute
Regular Function Symbol	::=	$\ldots$
Synthetic Function Symbol	::=	$\texttt{said}_d$: Info $\to$ Speech
	$\mid$	$\texttt{tdOn}_d$: Info $\to \mathrm{Attribute}$
	$\mid$	`canActAs`: Principal $\to \mathrm{Attribute}$
	$\mid$	`canSpeakAs`: Principal $\to \mathrm{Attribute}$
	$\mid$	$+$: Info $\times$ Info $\to$ Info
	$\mid$	$\mathcal{I}$: (Regular $\times$ Attribute) $\cup$ (Principal $\times$ Speech) $\to$ Info
	$\mid$	`exists`: Attribute
	$\mid$	$\ldots$
Substrate Relation Symbol	::=	`regcomp` : Regular$\times$Synthetic $\mid$ $\ldots$
Superstrate Relation Symbol	::=	`ensues` : Info $\times$ Info
	$\mid$	$\texttt{knows}_d$: Principal$\times$Info
	$\mid$	$\texttt{saysto}_d$: Principal$\times$Info$\times$Principal

Figure 6. DKAL vocabulary

A subexpression s of t is a *regular component* of t if s is regular, t is synthetic, and s is a leaf of syntree(t).

The substrate always has the following synthetic functions called *house constructors*: unary functions `said`, $\texttt{said}_0$, `tdOn`, $\texttt{tdOn}_0$, `canActAs`, `canSpeakAs`, binary functions $+$ and $\mathcal{I}$, and a constant `exists`; Fig. 6 gives their types.

Convention 8.1. Function symbols `said` and `tdOn` can be written as $\texttt{said}_\infty$ and $\texttt{tdOn}_\infty$ respectively. Thus $\texttt{said}_d$ denotes `said` when $d = \infty$ and denotes $\texttt{said}_0$ when $d = 0$, and similarly for $\texttt{tdOn}_d$. In the case of functions $\texttt{said}_d$, $\texttt{tdOn}_d$, `canActAs` and `canSpeakAs`, we write the function name of the house constructor followed by the argument, with no parentheses. For example, `canActAs Bob` is the attribute obtained by applying the function `canActAs` to the constant `Bob`. We write $x + y$ instead of $+(x, y)$. In the case of the function $\mathcal{I}$ we generally omit the function name altogether writing just `Bob is a user` rather than $\mathcal{I}$(`Bob, is a user`). For example, the full version of double rule TrustApp in Fig. 5 is

$$x \le +\big(\mathcal{I}(p, \texttt{said(x)}), \mathcal{I}(p, \texttt{tdOn(x)})\big)$$
$$x \le +\big(\mathcal{I}(p, \texttt{said}_0\texttt{(x)}) + \mathcal{I}(p, \texttt{tdOn}_0\texttt{(x)})\big)$$

The functions `canActAs` and `canSpeakAs` may be used for assignment of roles; their precise meaning is given by house rule Role in Fig. 7. To give a quick example, if it is known that p `canActAs Director`, and `Director canRead foo`, then it follows that p `canRead foo`. In the other direction, if it is known that p `canSpeakAs Director`, and p `said A isHired`, then it follows that `Director said A isHired`. Uses of the other house functions have been demonstrated in earlier sections.

The substrate may have user-introduced regular functions and relations. It may have user-introduced synthetic functions with values of type Attribute or Info. The only functions with values of type Speech are house constructors `said` and $\texttt{said}_0$.

Remark 8.2. A priori, one may expect that $\texttt{said}_d$ and $\texttt{tdOn}_d$ are relation of type Principal $\times$ Info, and `canActAs, canSpeakAs` are relations of type Principal $\times$ Principal. Why do we treat them as functions? First, there is a substantial increase of the expressivity of the language. Contrary to relations, functions can be nested. In particular, we can express quotations, like p `said` q `said` x. Second, DKAL communication is targeted. What would a proposition p `said` x mean? Who's the target of that communication? Third, there is in general no central authority in a distributed situation. What would a proposition p `tdOn` x mean? Who trusts p on x? From our point of view, phrases like p `said` x are infons, not propositions. We will revisit the issue in Remark 8.4.

8.2. Superstrate

There are five superstrate relations: `says, knows,` $\texttt{knows}_0$`, saysto` and $\texttt{saysto}_0$; Fig. 6 gives their types. Again $d \in \{0, \infty\}$ and subscript ∞ may be omitted. We write p $\texttt{knows}_d$ x instead of $\texttt{knows}_d(p, x)$, and p $\texttt{says}_d$ x `to` q instead of $\texttt{saysto}_d(p, x, q)$. We write x `ensues` y or $x \le y$ instead of $\texttt{ensue}(x, y)$. (This use of the $\le$ symbol is a mere convenience and does not preclude the use of the symbol in the substrate.)

$$
\begin{array}{rcll}
p\ \texttt{knows}\ q\ \texttt{said}_d\ x & \leftarrow & q\ \texttt{says}_d\ x\ \texttt{to}\ p & \text{(Say2know)}\\
p\ \texttt{knows}\ x & \leftarrow & p\ \texttt{knows}_0\ x & \text{(K0}\infty\text{)}\\
x \le\ x & & & \text{(EOrder)}\\
x \le\ z & \leftarrow & x \le y, y \le z & \\
x \le\ x+y & & & \text{(ESum)}\\
y \le\ x+y & & & \\
x+y \le z & \leftarrow & x \le z, y \le z & \\
t\ \texttt{exists} \le x & \leftarrow & t\ \texttt{regcomp}\ x & \text{(Exists)}\\
p\ \texttt{said}\ x \le\ p\ \texttt{said}_0\ x & & & \text{(Said0}\infty\text{)}\\
p\ \texttt{said}_d\ (x+y) \le p\ \texttt{said}_d\ x + p\ \texttt{said}_d\ y & & & \text{(SaidSum)}\\
p\ \texttt{said}_d\ x \le p\ \texttt{said}_d\ p\ \texttt{said}_d\ x & & & \text{(SelfQuote)}\\
p\ \texttt{tdOn}_d\ x \le p\ \texttt{tdOn}_d\ p\ \texttt{tdOn}_d\ x & & & \text{(Del}^-\text{)}\\
p\ \textit{attribute} \le q\ \textit{attribute} + p\ \texttt{canActAs}\ q & & & \text{(Role)}\\
q\ \textit{speech} \le p\ \textit{speech} + p\ \texttt{canSpeakAs}\ q & & &
\end{array}
$$

Figure 7. House Rules, part II

Remark 8.3. One can develop DKAL without relations $\texttt{knows}_d$, representing $p\ \texttt{knows}_d\ x$ with $p\ \texttt{says}_d\ x\ \texttt{to}\ p$. We choose to make knowledge explicit because of the fundamental role of knowledge and because the separation of knowing and saying is convenient technically as well.

The superstrate relations are computed over the substrate by the logic program that consists of the house rules in Figures 5 and 7 as well as of the rules given by assertions placed by principals. Assertions forms and the rules that assertions give rise to are described in the next subsection.

Remark 8.4. In Remark 8.2, we gave some reasons for making constructs $\texttt{said}_d$, $\texttt{tdOn}_d$, etc. functions and thus "pushing" them into the substrate. Having these constructs as substrate functions has one additional advantage. By means of house rules, we can impose natural axioms on these constructs. But one has to be careful if one is determined to keep query evaluation terminating and feasible.

8.3. Assertions

There are two forms of DKAL assertions:

1. $\texttt{A} :_d \quad x \qquad \leftarrow \quad x_1, \ldots, x_n,\ \textit{con},$
2. $\texttt{A} :_d \quad x\ \texttt{to}\ p \quad \leftarrow \quad x_1, \ldots, x_n, \textit{con}.$

Here A in both forms is a ground principal expression denoting the *owner* of the assertion; d is either ∞ or 0, and ∞ is typically omitted; $x, x_1, \ldots, x_n$ are infon expressions; and *con* is a *substrate constraint*, that is a conjunction of possibly-negated atomic formulas in the substrate vocabulary. All variables are regular, that is of type Regular; p is a principal variable called the *target variable*. Assertion 1 is a *knowledge assertion*. It does not have a target variable, and it gives rise to rule

$$
\begin{array}{l}
\texttt{A knows}_d\ x \leftarrow\\
\quad \texttt{A knows}_d\ x_1, \ldots, \texttt{A knows}_d\ x_n,\\
\quad \texttt{A knows}_d\ t_1\ \texttt{exists}, \ldots,\\
\quad \texttt{A knows}_d\ t_k\ \texttt{exists},\ \textit{con}
\end{array}
$$

where the list $t_1, \ldots, t_k$ consists of the variables in $x, x_1, \ldots, x_n$, *con* and of the non-ground regular components of assertion head x. Since the rule contains the conditions $\texttt{A knows}_d\ t_i\ \texttt{exists}$, we say that $t_1, \ldots, t_k$ are $\texttt{A}$-*bounded* in the assertion. Assertion 2 is a *speech assertion* and gives rise to rule

$$
\begin{array}{l}
\texttt{A says}_d\ x\ \texttt{to}\ p \leftarrow\\
\quad \texttt{A knows}_d\ x_1,\ \ldots,\ \texttt{A knows}_d\ x_n,\\
\quad \texttt{A knows}_d\ t_1\ \texttt{exists}, \ldots,\\
\quad \texttt{A knows}_d\ t_k\ \texttt{exists},\ \textit{con}
\end{array}
$$

where the list $t_1, \ldots, t_k$ consists of the variables of the assertion and the non-ground regular components of the assertion head x, with the exception of the target variable p.

Semantically there is no difference between an assertion and the rule that it gives rise to. The difference is purely syntactical. Assertions provide simpler and more convenient way to write rules.

Note that any assertion rule conditions its head only on the knowledge of the assertion owner possibly augmented with a substrate constraint; this is key in dealing with the information leakage problem in §4. In case $d = 0$, the knowledge is internal; that property, inherited from SecPAL, is key in delegation restriction §6.

8.4. Queries

Fix a substrate X and let Υ be the vocabulary of X extended with the superstrate relation names. Further, consider an authorization policy (that is a set of assertions) $\mathcal{A}$ in the vocabulary Υ. Let Π be the logic program that consists of the house rules and the assertions in $\mathcal{A}$. And let $\Pi(X)$ be the *state of knowledge* determined by X and Π, that is the enrichment of X by means of superstrate relations computed by Π over X.

A *basic query* in vocabulary Υ is a formula $p\ \texttt{knows}_d\ t(v_1,\ldots,v_k)$ where p is a ground principal expression in the substrate vocabulary, t is an infon expression with variables $v_1,\ldots,v_k$, the variables are all regular, and d is 0 or ∞. The query is evaluated over the state of knowledge $\Pi(X)$. The *answer* is the set of tuples $\langle b_1,\ldots,b_k\rangle$ of regular elements of X such that the type of b_i is that of v_i and

$$\begin{aligned}\Pi(X) \models\ & p\ \texttt{knows}_d\ t(b_1,\ldots,b_k) \wedge \\ & p\ \texttt{knows}_d\ b_1\ \texttt{exists}\ \wedge \cdots \wedge \\ & p\ \texttt{knows}_d\ b_k\ \texttt{exists}.\end{aligned}$$

For any ground principal expression p, a *p-centric query* is a first-order formula. We define p-centric queries inductively.

1. Every substrate constraint is a p-centric query.
2. Every basic query $p\ \texttt{knows}_d\ t(v_1,\ldots,v_k)$ is p-centric.
3. If Q_1 and Q_2 are p-centric queries then so are $\neg Q_1$, $Q_1 \wedge Q_2$ and $Q_1 \vee Q_2$.
4. If $Q(v)$ is a p-centric query then so are formulas
$\exists v\ \big((p\ \texttt{knows}\ v\ \texttt{exists})\ \wedge\ Q(v)\big)$,
$\forall v\ \big((p\ \texttt{knows}\ v\ \texttt{exists})\ \longrightarrow\ Q(v)\big)$.

It follows that all quantifications in a p-centric query are restricted to elements known to p. The *answer* to a p-centric query is defined by induction, in the obvious way.

In particular, a Boolean combination of substrate constraints and p-centric basic queries is a p-centric query. The availability of negations in queries can be used for conflict resolution at the decision point. For example, in a deny-override system, with read guard RG, read access to File 13 would be given to the users in the answer to the query:
$\texttt{RG knows}\ p\ \texttt{hasReadAccessTo File 13} \wedge$
$\neg\ \big(\texttt{RG knows}\ p\ \texttt{deniedAccessTo File 13}\big)$.

In this paper, a *query*, that is DKAL query, is a p-centric query for some p.

9. Query evaluation

The flexible use of functions makes DKAL closer to Prolog than to Datalog. It is of course only too easy to write a non-terminating program in Prolog. But DKAL is carefully calibrated to ensure the termination of an algorithm that computes answers to queries.

Recall that state elements split into regular and synthetic. In policy assertions, variables range over regular elements only. Further, consider any assertion α, and let $\texttt{A}$ be the owner of α and t be a variable in α or a non-ground regular component of α's head. Unless α is a speech assertion and t is the target variable, we require that α contains condition $\texttt{A}\ \texttt{knows}_d\ t\ \texttt{exists}$ for the appropriate d; see 8.3. In particular all non-target variables of α's head occur in the body. Also, for the purpose of evaluating the body of α, the relevant values of non-target variables are those whose existence is known to $\texttt{A}$. The requirement is a semantic safety condition that prevents the knowledge of $\texttt{A}$ from exploding.

The infon $t\ \texttt{exists}$ carries no information about t except that t exists. See rule Exists in Fig. 7 in this connection. Relation $\texttt{regcomp}$ was defined in §8.1. The intuitive meaning of relation $t\ \texttt{regcomp}\ x$ is that t appears in x in an essential way. By rule Exists, infon $t\ \texttt{exists}$ is the least informative among infons that mention t in an essential way.

The semantic safety condition allows us to show that only the regular elements that are explicitly mentioned in the policy are relevant and need to be considered as possible values for variables when evaluating the policy. Since the policy is finite, the number of relevant regular elements is finite.

Things are much more involved with synthetic elements. While assertions have only regular variables, many house rules of DKAL have variables ranging over synthetic elements, in particular over infons. We prove that the number of synthetic elements needed to evaluate a given query under a given policy is finite. The proof is elaborate and uses the nature of the DKAL house rules; it is written in full details in the technical report [15].

We construct a query evaluation algorithm that takes advantage of the fact that only finitely many regular and synthetic elements are relevant.

Theorem 9.1. *For any substrate X, given an authorization policy $\mathcal{A}$ over X and a query Q, the DKAL query evaluation algorithm computes the answer to Q under $\mathcal{A}$ and X.*

The proof is elaborate and long, even for an appendix of an extended abstract. It is written in full details in the technical report [15]. Fortunately one does not need to know the proof in order to use the algorithm. Note that the answer is always finite.

10. Worst case complexity

Fix a substrate X. To simplify the complexity analysis of the query evaluation algorithm, we assume that the algorithm Eval of §8.1 works in constant time: given a function or relation symbol S and the appropriate tuple $\bar{a}$ of the elements of X, Eval evaluates $S(\bar{a})$ in constant time. Essentially we count only the number of Eval calls and ignore Eval's computation time. Alternatively we could make a natural assumption that Eval works in time bounded by a polynomial of the maximal length of an input string. That polynomial would have to be taken into account in the following theorem but would not affect our results in any essential way. The analysis of Eval is orthogonal to the main issue of this theorem.

If β is an assertion or a query, then the *quotation depth* of β is the depth to which `said` and $\texttt{said}_0$ (possibly mixed) are nested in β, and the *width* of β is the number of variables in β. Note that the width is zero in the ubiquitous case of basic yes/no queries.

Theorem 10.1. *The time that the query evaluation algorithm needs to answer a query Q under an authorization policy $\mathcal{A}$ is bounded by a polynomial in*

$$(\text{length}(\mathcal{A}) + \text{length}(Q))^{\delta+1+w},$$

where δ (resp. w) bounds the quotation depth (resp. the width) of the policy assertions and of the query. In the important case where δ and w are fixed, the computation time is polynomial in $\text{length}(\mathcal{A}) + \text{length}(Q)$.

A detailed proof is found in the technical report [15]. The time bound can be sharpened but the importance of the worst case need not be exaggerated. Typical cases seem to be much different.

11. SecPAL to DKAL translation

In this section, SecPAL is the language defined in [4, 5] and not an implementation of the language. We presume, without loss of generality, that the names of sorts, functions and relations introduced explicitly in §8 do not occur in SecPAL.

We describe a natural translation τ of SecPAL into a version of DKAL, called Open DKAL, obtained by augmenting DKAL with double rules

$$p\ \texttt{says}_d\ x \leftarrow p\ \texttt{knows}_d\ x \qquad \text{(O1)}$$

$$p\ \texttt{knows}_d\ x \leftarrow p\ \texttt{says}_d\ x \qquad \text{(O2)}$$

Here $p\ \texttt{says}_d\ x$ is an Open DKAL abbreviation for $p\ \texttt{says}_d\ x\ \texttt{to}\ q$ where q is any fresh variable. O1 reflects the all-knowledge-is-common nature of SecPAL. O2 is a mere convenience; it allows us to translate SecPAL's `says` by DKAL's `says` rather than by DKAL's `knows`. In the rest of this section, by default, DKAL is Open DKAL.

Let CD be a constraint domain of SecPAL. We view CD as a first-order structure. We turn CD into a DKAL substrate X whose regular elements are precisely the elements of CD. X extends CD in two ways, by expanding the vocabulary and adding synthetic elements. We define $\tau N = N$ for every name in the CD vocabulary. X has the built-in DKAL functions. In addition, for each SecPAL predicate *pred*, X has a synthetic function, also denoted *pred*, taking attribute values; the domain of the new function *pred* is the domain of the original SecPAL predicate *pred*. X has no other sorts, functions or relations.

Fig. 8 completes the definition of the translation τ. The lines in the figure correspond to the SecPAL grammar [4]. In the final line, tagged Assertion, *all* is a fresh variable not occurring in the assertion, and d takes values $0, \infty$. SecPAL verbphrases become DKAL attributes, SecPAL facts become DKAL infons, and each SecPAL assertion gives rise to two DKAL assertions, one with $d = 0$ and the other with $d = \infty$. In SecPAL, an assertion context AC is a set of assertions. Accordingly an assertion context AC composed of n SecPAL assertions gives rise to a DKAL policy $\tau(\text{AC})$ composed of $2n$ assertions.

Theorem 11.1 (Embedding Theorem)**.** *Let AC be a safe SecPAL assertion context, and let Π be the logic program composed of the policy $\tau(AC)$ and the Open DKAL house rules. If*

$$AC, d \vdash A\ \texttt{says}\ f$$

in SecPAL, where A is a principal constant, f is a ground flat fact expression, and $d \in \{0, \infty\}$, then

$$\Pi(\text{Sub}) \models A\ \texttt{says}_d\ \tau f$$

in Open DKAL

Remark 11.2. It is possible to translate SecPAL to the original DKAL rather than Open DKAL. We mentioned already that double rule O2 is not essential for translation. The necessary instances of double rule O1 can be incorporated into the translation of SecPAL assertions. In SecPAL, a fact is *flat* when it does not contain `can say`, and every fact f has the form $e_1\ \texttt{can say}_{d_1} \ldots e_n\ \texttt{can say}_{d_n}\ g$ where $n \geq 0$ and g is flat. We refer to g as the *flat seed* of f. We refer to each of the facts $e_{k+1}\ \texttt{can say}_{d_{k+1}} \ldots e_n\ \texttt{can say}_{d_n}\ g$, $0 \leq k \leq n$, as a *subfact* of f. To translate into the original DKAL, define $\tau(A\ \texttt{says}\ f\ \texttt{if}\ f_1, \ldots, f_n, \mathit{con})$ to be the set of the following DKAL assertions, instead of the ones at Assertion in Fig. 8:

$$A_d : \tau(f)\ \leftarrow \tau(f_1), \ldots, \tau(f_n), \mathit{con}$$
$$A_d : \tau(f')\ \texttt{to}\ \mathit{all} \leftarrow \tau(f')$$

$$
\begin{aligned}
\tau(e) &= e && \text{(Variable, constant)}\\
\tau(\mathit{con}) &= \mathit{con} && \text{(Constraint)}\\
\tau(\mathit{pred}) &= \mathit{pred} && \text{(Predicate)}\\
\tau(\mathit{pred}\ e_1\ \dots\ e_n) &= \mathit{pred}(e_1, \dots, e_n) && \text{(Verbphrase)}\\
\tau(\texttt{can say}_d\ \mathit{fact}) &= \texttt{tdOn}_d\ \tau(\mathit{fact}) &&\\
\tau(\texttt{can act as}\ e) &= \texttt{canActAs}\ e &&\\
\tau(e\ \mathit{verbphrase}) &= \mathcal{I}(e, \tau(\mathit{verbphrase})) = e\ \tau(\mathit{verbphrase}) && \text{(Fact)}\\
\tau(\texttt{A says}\ \mathit{fact}\ \texttt{if}\ \mathit{fact}_1, \dots, \mathit{fact}_n, \mathit{con}) &= \texttt{A :}_d \quad \tau(\mathit{fact})\ \texttt{to}\ \mathit{all} \leftarrow \tau(\mathit{fact}_1), \dots, \tau(\mathit{fact}_n), \mathit{con} && \text{(Assertion)}
\end{aligned}
$$

Figure 8. The SecPAL-2-DKAL translation map τ

where f' ranges over the subfacts of f. The obvious analog of the embedding theorem holds but the proof is a bit more involved.

Proposition 11.3. *The converse of the embedding theorem is not true. There is an assertion context AC and a SecPAL query* A `says` f *such that* $\Pi(\text{Sub}) \models A$ `says` $\tau(f)$ *but* $AC, \infty \not\vdash A$ `says` f.

If one weakens DKAL by removing rules Del and Del$^-$, then the embedding theorem survives and its converse holds too. We do not see Proposition 11.3 as a drawback of Open DKAL. Moreover, it is advantageous if more justified requests get positive answers. Note also that only justified requests get positive answers in SecPAL, DKAL and Open DKAL, and that Open DKAL justifications are every bit as convincing as those of SecPAL.

One may wonder why the rules Del$^-$ and Del are in DKAL in the first place. The question is natural in the context of SecPAL and Open DKAL. The small additional expressivity may not justify the two rules. But the situation is different in DKAL proper. The open character of SecPAL makes delegation easy. If p `says` q `can say` x and if q `says` x then p `says` x. This is elegant and simple. But it also opens too large a venue for p to find out (possibly using a probing attack) what q says. Targeted communication and confidentiality complicate delegation in a substantial way. It should be possible to exercise the delegated authority in such a way that the delegator is not involved and cannot discover by probing whether the delegated authority was exercised and by whom. Del naturally captures the intuitive meaning of the delegatability of trust. The other rule, Del$^-$, is a minor issue.

12. Related work

The literature on the use of logic in authorization policies for decentralized system is too rich to be fairly reviewed in few paragraphs. We concentrate mostly on recent work that is more closely related to this paper.

Speaks-For Calculus [1] pioneered the use of logic (a form of modal logic in the Speaks-For case). In particular Speaks-For addressed delegation and representation, and introduced the `says` construct that, in one form or another, has been popular in authorization literature ever since. Datalog possibly with constraints [20] was the foundation for many later logic-based authorization languages. DKAL resurrected an attraction of Speaks-For absent in the existing Datalog based literature (as far as we know): nested quotations.

The term "trust management" was coined in [9]. The article introduced PolicyMaker that later evolved into KeyNote [10]. KeyNote expresses involved scenarios. In particular, it allows a principal to delegate a subset of his rights to another principal, and it has thresholds. But there are common authorization scenarios that cannot be expressed in KeyNote. Typically they involve a right granted on the basis of an attribute that originates from a lateral source; see the introduction to [19] in this connection.

Our late genealogy consists primarily of Datalog based languages Binder [12], Delegation Logic [18, 19] and SecPAL [5, 13]. Binder [12] extends Datalog with an import/export construct `says` that connects Datalog programs maintained by different principals and makes the issuer of an imported assertion explicit. A principal A may for example condition a Datalog rule on B `says` `employee`(C, E). The rule will fire when principal B exports `employee`(C, E). A may express his trust in B on `employee`(x, y) by means of a Datalog rule `employee`(x, y) `:-` B `says` `employee`(x, y). Recently the import and export aspects of Binder's `says` were separated in SeNDlog [2]. This useful advance, inspired by a database query language NDlog [23], and DKAL's targeted communication attend to the same concern but were independent.

Delegation Logic features vocabulary specifically designed for authorization policies. Contrary to Binder, it does not have explicit issuers for all assertions. But it has a number of useful, authorization specific constructs, including ones for delegations, representations, and thresholds. For the purpose of execution, Delegation Logic is reduced to Datalog.

SecPAL has both explicit issuers of assertions and specific constructs designed with distributed systems authorization policy in mind. The number of constructs is deliberately kept low, but the language is expressive and captures many standard authorization scenarios, including discretionary and mandatory access control, role hierarchies, separation of duties, threshold-constrained trust, attribute-based delegation, and delegation controlled by constraints, depth, and width, see [5, Section 5]. The semantics of the language is defined directly, using a few very condensed deduction rules. For the purpose of execution, SecPAL is reduced to Datalog with constraints. Nested $\texttt{can say}_0$ facts are used for bounded depth delegation, and the SecPAL deduction laws give rise to semantics that prevents any circumventing of the delegation bound.

While its authors see SecPAL as Datalog based [3], we find it more illuminating to see SecPAL from the point of view of existential fixed-point logic sketched in §2. From that point of view, `can say` (a.k.a. $\texttt{can say}_\infty$) and $\texttt{can say}_0$ are fact-valued functions that can be nested in SecPAL. SecPAL policies are reduced to safe Constraint Datalog programs by converting nested $\texttt{can say}_d$ facts to relations of arity dependent on the nesting depth, which is finite in any given policy and can only decrease in deductions.

The RT family languages [21] are also Datalog based. The languages have roles instead of attributes, and principals may condition membership in a role they control on membership in roles controlled by other principals. Both RT and SecPAL extend Datalog with constraints. In RT tractability with constraints is obtained by assuming that the constraint domain satisfies quantifier elimination. SecPAL uses instead a syntactic safety condition that guarantees that constraint variables are instantiated at the time of evaluation.

13. Conclusion and future work

We designed an authorization language DKAL which exceeds the expressivity of previous languages in the literature in a number of ways and yet maintains feasible complexity bounds for answering authorization queries. The language has several innovative features including these: targeted communication and a distinction between knowing and saying; quotations that can be nested and, more generally, flexible formation of expressions with unrestricted use of functions that can be nested and mixed; extended use of an underlying substrate structure that may be very rich; stronger delegation semantics; and an information order that makes the language more succinct and comprehensible.

Policies written in SecPAL [5], a recent expressive authorization language, can be translated into DKAL, and so all SecPAL expressible authorization scenarios are also expressible in DKAL. We attempted to illustrate the usefulness of the new features of DKAL for user-centric scenarios, prevention of information leakage, abstraction of cryptographic protocols, and design of more modular distributed authorization policies. Our algorithm for answering DKAL queries has worst-time complexity within the SecPAL time complexity bounds.

The DKAL query answering algorithm is currently implemented in Prolog. We work toward more substantial implementation and future deployment of DKAL. There are several appetizing directions to expand DKAL. One is to develop syntax and semantics for targeting assertions. Currently only targeting of infons is expressible in DKAL. Another direction is related to house rules. One may want to enrich them with additional ensue rules. The problem is to understand which ensue rules can be added without violating the time complexity results. Yet another direction is to allow the policies to use negation in a stratified way.

References

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin, "A Calculus for Access Control in Distributed Systems," ACM Transactions on Programming Languages and Systems, 15:4, 706–734, 1993.

[2] Martín Abadi and Boon Thau Loo, "Towards a Declarative Language and System for Secure Networking", in International Workshop on Networking Meets Databases (NetDB '07), 2007.

[3] Private communication, January 2008.

[4] Moritz Y. Becker, Cédric Fournet and Andrew D. Gordon, "SecPAL: Design and Semantics of a Decentralized Authorization Language", Technical Report MSR-TR-2006-120, Microsoft Research, September 2006.

[5] Moritz Y. Becker, Cédric Fournet and Andrew D. Gordon, "SecPAL: Design and Semantics of a Decetralized Authorization Language", 20th IEEE Computer Security Foundations Symposium (CSF), 3–15, 2007.

[6] Moritz Y. Becker and Peter Sewell, "Cassandra: Distributed Access Control Policies with Tunable Expressiveness", in IEEE 5th International Workshop on Policies for Distributed Systems and Networks, 159-168, 2004.

[7] Moritz Y. Becker and Peter Sewell, "Cassandra: Flexible Trust Management, Applied to Electronic Health Records", in IEEE Computer Security Foundations Workshop, 139-154, 2004.

[8] Andreas Blass and Yuri Gurevich, "Existential Fixed-Point Logic", Springer Lecture Notes in Computer Science 270 (1987), 20–36. Available at `http://research.microsoft.com/~gurevich/Opera/73.pdf`.

[9] Matt Blaze, and Joan Feigenbaum, and Jack Lacy, "Decentralized Trust Management", in Proc. 1996 IEEE Symposium on Security and Privacy, 164–173, 1996.

[10] Matt Blaze, Joan Feigenbaum, Angelos D. Keromytis, "The Role of Trust Management in Distributed Systems Security", in Secure Internet Programming, 185–210, 1999.

[11] S. Cantor, J. Kemp, R. Philpott, and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005. `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`.

[12] John DeTreville, "Binder, a Logic-Based Security Language", in IEEE Symposium on Security and Privacy, 105-113, 2002.

[13] Blair Dillaway, "A Unified Approach to Trust, Delegation, and Authorization in Large-Scale Grids," Technical Paper, Microsoft Corporation, September 2006.

[14] Herbert Enderton, "A Mathematical Introduction to Logic", 2nd edition, Academic Press.

[15] Yuri Gurevich and Itay Neeman, "DKAL: Distributed Knowledge Authorization Language", MSR-TR-2007-116 (August 2007) and MSR-TR-2008-09 (January 2008 Revision), Microsoft Research.

[16] Wilfrid Hodges, "Model Theory", Cambridge University Press, 1993.

[17] Joxan Jaffar and Michael J. Maher, "Constraint logic programming: A survey," Journal of Logic Programming 19/20 (1994), 503-580.

[18] Ninghui Li, "Delegation Logic: A Logic-Based Approach to Distributed Authorization", Ph.D. thesis, New York University, September 2000.

[19] Ninghui Li, Benjamin N. Grosof and Joan Feigenbaum, "Delegation Logic: A Logic-Based Approach to Distributed Authorization", ACM Trans. on Information and System Security (TISSEC) 6:1 (February 2003), 128–171.

[20] Ninghui Li and John C. Mitchell, "Datalog with Constraints: A Foundation for Trust Management Languages", PADL 2003, V. Dahl and P. Wadler (Eds.), LNCS 2562, 58–73, 2003.

[21] Ninghui Li and John C. Mitchell, "RT: A Role-Based Trust-Management Framework", in Proceedings of the Third DARPA Information Survivability Conference and Exposition (DISCEX III), 201–212, April 2003.

[22] Ninghui Li, William H. Winsborough, and John C. Mitchell, "Beyond Proof-of-Compliance: Safety and Availability Analysis in Trust Management", in Proceedings of 2003 IEEE Symposium on Security and Privacy, 123–139, May 2003.

[23] B.T. Loo, T. Condie, M. Garofalakis, D.E. Gay, J.M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative Networking: Language, Execution and Optimization." In Proceedings of ACM SIGMOD International Conference on Management of Data (June 2006).

[24] Yuri Matiyasevich, *Hilbert's Tenth Problem,* MIT Press, 1993.

[25] Elliott Mendelson, "Introduction to Mathematical Logic," 4th edition, Academic Press.

[26] "Prolog," Wikipedia. `http://en.wikipedia.org/wiki/Prolog`.

[27] OASIS. Security Assertion Markup Language (SAML). `www.oasis-open.org/`.

[28] Oxford English Dictionary, 2nd edition, Oxford University Press, 1989.

[29] Alfred Tarski, "A Lattice-Theoretical Fixpoint Theorem and its Applications", *Pacific Journal of Mathematics* 5:2 (1955), 285–309.