# CSchema: A Downgrading Policy Language for XML Access Control

Dong-Xi Liu (刘东喜)

*Department of Mathematical Informatics, University of Tokyo, Japan*

E-mail: liu@mist.i.u-tokyo.ac.jp

Received October 15, 2005; revised July 2, 2006.

**Abstract** The problem of regulating access to XML documents has attracted much attention from both academic and industry communities. In existing approaches, the XML elements specified by access policies are either accessible or inaccessible according to their sensitivity. However, in some cases, the original XML elements are sensitive and inaccessible, but after being processed in some appropriate ways, the results become insensitive and thus accessible. This paper proposes a policy language to accommodate such cases, which can express the downgrading operations on sensitive data in XML documents through explicit calculations on them. The proposed policy language is called *calculation-embedded schema* (CSchema), which extends the ordinary schema languages with *protection type* for protecting sensitive data and specifying downgrading operations. CSchema language has a type system to guarantee the type correctness of the embedded calculation expressions and moreover this type system also generates a security view after type checking a CSchema policy. Access policies specified by CSchema are enforced by a validation procedure, which produces the released documents containing only the accessible data by validating the protected documents against CSchema policies. These released documents are then ready to be accessed by, for instance, XML query engines. By incorporating this validation procedure, other XML processing technologies can use CSchema as the access control module.

**Keywords** access control, programming language, security policy, type system, XML

## 1 Introduction

XML is the standard data format for exchanging information on the Internet. In many cases, it is desirable to allow different users to access different parts in one document according to their privileges or roles. This is the problem of XML access control, which has attracted many research efforts to solve it. For example, some approaches[1−6] and standards[7,8] have been proposed.

Access control mechanism is always an integral part for information systems, in which access policy is used to specify the accessibility of the sensitive resources to different users. For example, the system security policy in UNIX systems can grant or deny access to files to users. The existing approaches to XML access control also follow this style. Their policy languages specify which elements in XML documents can be accessed and which cannot probably under some conditions[9].

This style is suitable for protecting physical objects, such as files or directories, but not necessarily suitable for protecting data stored in XML documents. For XML data protection, the released data sometimes needs to be *computed* from the original sensitive data, not directly selected from them. That is, the original sensitive data is not accessible, but the data computed from them is, as shown by the following motivating example. From the perspective of information flow security[10], this case asks downgrading of the sensitive data from high security level (inaccessible) to low security level (accessible).

### 1.1 Motivating Example

Consider the XML file in Fig.1(a), which stores the staff information for a company.

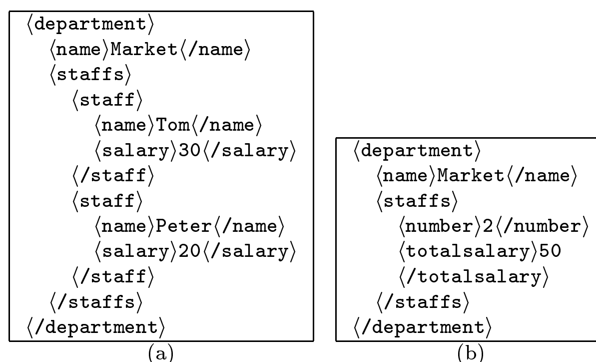


Fig.1. Example XML file. (a) Original data. (b) Data for the assistant.

The users of this file include all staffs in the department and a financial assistant. For some reason, the company needs to impose the following access policy:

- Tom and Peter can access the whole document except for each other's salary;
- the assistant is authorized to see a document in Fig.1(b), which is *computed* from the original data.

The document for the assistant includes two computed elements: the element `number` for the number of staffs and the element `totalsalary` for the sum of Tom and Peter's salary. In other words, the assistant is not allowed to know who is in the department and his/her salary.

The current approaches cannot specify such kind of policies because they are only able to *hide* inaccessible elements or *release* accessible elements existing in the

Regular Paper

Supported by the Program "Comprehensive Development of e-Society Foundation Software" of the MEXT, Japan, under the project "Programmable Structured Documents".

protected document, but cannot construct new accessible elements.

### 1.2 CSchema Overview

In this paper, programming language technologies, mainly types and operational semantics, are used to provide a novel way of enforcing flexible XML access control. The proposed policy language is called *calculation-embedded schema* (CSchema), which extends the ordinary schema languages, such as XML Schema or DTD, with *protection type*. This type consists of three components: the type of sensitive data in the original document, the type of the data accessible to users, and an expression to compute the accessible data. This expression is the embedded calculation in schema and describes the downloading operation. The intuitive meaning of protection type is that the sensitive data will be replaced by the data computed by the embedded expression. Since CSchema policies include embedded calculations, we provide a type system to guarantee their type correctness, and after type checking, this type system also generates a security view[1] with respect to the checked CSchema policy.

Writing a CSchema policy is just similar to writing an ordinary schema for XML documents, except that when meeting with a type that corresponds to sensitive data, the type should be changed into a *protection type*. Some general XML processing languages, like Java or CDuce[11], can transform the original document to the released document shown in Fig.1, but writing policy with them is inconvenient since they cannot let policy writers focus on the sensitive data, explained more by the example in Section 6.

The access policy expressed in CSchema is enforced by an extended validation procedure. XML is an external format to represent data, and by validating an XML file against a schema, either an internal representation of the file is generated or the validation fails[12]. In CSchema, when validating an external value against a protection type, instead of really performing validation as usual, the expression in this type will be evaluated and its result will be used as the internal value at the position of this external value, and thus this external value is hidden. If a document does not include sensitive data, then the CSchema policy for this document does not have any protection type, so the enforcement of this policy does not incur any overhead for access control purpose since it just performs the ordinary validation. The implementation of this enforcement method needs only a conservative extension to the ordinary validation procedure by supporting validation against protection type.

The contributions of this work are summarized as follows.

• Motivating the downgrading problem for XML access control, which moves the focus of downgrading policies from the traditional code level to data level.

• Designing a calculation-embedded schema language to express downgrading access policies for XML access control.

• Formalizing a type system to check the type correctness of CSchema policies and automatically generate security views.

• Enforcing CSchema policy by validation that can be implemented by a conservative extension to the ordinary validation procedure.

The remainder of this paper is organized as follows. Section 2 gives the syntax of CSchema and represents the motivating example by this syntax. In Section 3, the type system for CSchema is formalized and illustrated. Section 4 shows the validation procedure together with the dynamic semantics of embedded expressions and gives a property about CSchema. Section 5 discusses another potential application of CSchema and compares it with other approaches. Section 6 surveys the related work. Section 7 concludes this paper.

## 2 Syntax of CSchema

The syntax of CSchema is presented in Fig.2. It includes two parts: the syntax of types and the syntax of expressions. A CSchema policy is a type $\tau$.

$$
\begin{array}{lcl}
k & ::= & \star \mid \star \mapsto \star \\
\tau & ::= & t \mid () \mid \texttt{string} \mid \texttt{int} \mid \texttt{bool} \mid \langle l \rangle [\tau] \\
 & & \mid \tau_1, \tau_2 \mid \tau_1 \mid \tau_2 \mid \tau * \mid \tau_1 \mapsto \tau_2 \& e \mid \mu t.\tau \\
e & ::= & d \mid /p \mid \texttt{self} :: l/p \mid f(e) \\
d & ::= & x \mid () \mid \texttt{isempty}(d) \mid s \mid d_1 =_{\mathrm{s}} d_2 \mid n \mid d_1 + d_2 \mid \texttt{true} \mid \texttt{false} \\
 & & \mid \texttt{if}\ d\ \texttt{then}\ d_1\ \texttt{else}\ d_2 \mid \langle l \rangle [d] \mid d/p \mid d_1, d_2 \mid \texttt{head}(d) \mid \\
 & & \texttt{tail}(d) \mid f(d) \\
p & ::= & \epsilon \mid \texttt{child} :: l/p \\
\texttt{G} & ::= & . \mid \Gamma, \texttt{fun}\ f(x : \tau_1) : \tau_2 = d
\end{array}
$$

Fig.2. Syntax of CSchema.

### 2.1 Syntax of Types

The syntactic categories $k$ and $\tau$ in Fig.2 are relevant to the syntax of types. The types given here are built upon regular expression types[13]. A type can be an atomic type including the type variable $t$, the empty sequence `()`, `string`, `int`, `bool`, or a composed type including the element type $\langle l \rangle [\tau]$, the sequence type $\tau_1$, $\tau_2$, the choice type $\tau_1 \mid \tau_2$, the type $\tau*$, the *protection type* $\tau_1 \mapsto \tau_2 \& e$ and the recursive type $\mu t.\tau$. The occurrence modifier $*$ means zero or more occurrences of the modified type. In this syntax, we omit other occurrence modifiers ? and +. Actually, they can be defined using the existing constructs: $\tau? = () \mid \tau$ for optional occurrence of $\tau$, and $\tau+ = \tau, \tau*$ for one or more occurrences of $\tau$. The element type $\langle l \rangle [\tau]$ is said to be the *parent* of $\tau$, and in turn $\tau$ is its *child*. Later, $b$ is often used to range over the set {`bool`, `int`, `string`, `()`}.

The novelty in this type language is the *protection type* $\tau_1 \mapsto \tau_2 \& e$, where $\tau_1$, called *original component*, is the type of the original sensitive data; $\tau_2$, called *view component*, is the type of the data that replaces the

sensitive data and is exposed to users; $e$ is the *embedded expression* for an downgrading operation to compute the released data. Hence, the expression $e$ must have the type $\tau_2$. This invariant is guaranteed by the type system in next section. For the access control purpose, the type $\tau_2$ in the *protection type* $\tau_1 \mapsto \tau_2 \& e$ should no longer be constructed directly or indirectly by any other *protection types* since it is the type for released data that need not be protected any more. This requires to distinguish between the types which include *protection types* and those which do not. As usual, a kind system, shown in Fig.3, is used to classify types. The judgment $\tau :: k$ means the type $\tau$ has the kind $k$. If $k$ is $\star$ then $\tau$ does not include *protection types*; if $k$ is $\star \mapsto \star$ then $\tau$ does. And $k_1 \oplus k_2$ is defined as: $\texttt{if } k_1 = \star \texttt{ and } k_2 = \star \texttt{ then } k = \star \texttt{ else } k = \star \mapsto \star$. That is, a composed type has kind $\star \mapsto \star$ only if one of its constituent types has such kind.

$$\frac{}{b :: \star} \quad \frac{}{t :: \star} \quad \frac{}{\tau_1 \mapsto \tau_2 \& e :: \star \mapsto \star} \quad \frac{\tau :: k}{\langle l \rangle [\tau] :: k}$$

$$\frac{\tau :: k}{\tau * :: k} \quad \frac{\tau :: k}{\mu t.\tau :: k} \quad \frac{\tau_1 :: k_1 \quad \tau_2 :: k_2}{\tau_1, \tau_2 :: k_1 \oplus k_2}$$

$$\frac{\tau_1 :: k_1 \quad \tau_2 :: k_2}{\tau_1 \mid \tau_2 :: k_1 \oplus k_2}$$

Fig.3. Kinding rules.

### 2.2 Syntax of Expressions

An expression $e$ is used only when defining a *protection type*. It can be a variable $d$, an absolute path $/p$, a relative path $\texttt{self::}l/p$ or a function application $f(e)$.

In Fig.2, $G$ contains the globally defined functions that can be used when writing expressions. Each function is defined in the form $\texttt{fun}\, f(x : \tau_1) : \tau_2 = d$, which takes the argument $x$ of the type $\tau_1$ and returns a value of the type $\tau_2$. The function body is in syntactic category $d$. By this syntax, the absolute or relative path expressions cannot be used in a function body, but they can be used as the argument in the function application $f(e)$. Recursive functions are supported by this syntax and they are useful to deal with values of recursive types.

The path expression, $/p$ or $\texttt{self::}l/p$, can be used to locate either the type components in CSchema policies at the type checking stage or the data in XML documents at the validating stage. Correspondingly, the context for evaluating $/p$ is a whole CSchema policy or a whole document, and the context for $\texttt{self::}l/p$ is the original component of the protection type containing it or the data validated against this protection type. The label $l$ in a path expression can be a string representing an element name or the symbol # representing basic values. In addition, each path expression ends with $\epsilon$.

The other expressions include variable $x$ as well as introduction and elimination forms of data of each type. For example, $\langle l \rangle [d]$ constructs an element and $d/p$ is to use it; $d_1, d_2$ constructs a sequence data and it can be destructed by $\texttt{head}(d)$ and $\texttt{tail}(d)$; $s$ represents a string, $n$ an integer; $d_1 =_s d_2$ is to judge whether two strings $d_1$ and $d_2$ are equal.

### 2.3 Examples

This section represents the motivating example in Section 1 by the syntax of CSchema. A schema for that document without access control can be represented as follows:

```
τ = ⟨department⟩[⟨name⟩[string],
        ⟨staffs⟩[⟨staff⟩[⟨name⟩[string],
                ⟨salary⟩[int]]*]].
```

The access policy for Tom is given in Fig.4, where a function `auth` is defined to remove salary information from its argument element if this element is not for Tom. In this policy, the element type *staff* is changed into a protection type. Its view component indicates that each released *staff* element may or may not contain child element *salary* and this element is the result of applying `auth` to the original *staff* element specified by the path $\texttt{self :: staff}/\epsilon$.

```
fun auth (x:⟨staff⟩[⟨name⟩[string],
      ⟨salary⟩[int]]):⟨staff⟩[⟨name⟩[string], ⟨salary⟩
      [int]?] =
    if x/child::name/child::#/ϵ = ‘‘Tom’’ then x
    else ⟨staff⟩[⟨name⟩[x/child::name/child::#/ϵ]]
τ = ⟨department⟩[⟨name⟩[string], ⟨staffs⟩[
      (⟨staff⟩[⟨name⟩[string], ⟨salary⟩[int]]↦
      ⟨staff⟩[⟨name⟩[string], ⟨salary⟩[int]?]&
      auth(self::staff/ϵ))*]]
```

Fig.4. Access policy for Tom.

The policy for the financial assistant is given in Fig.5. Two functions are defined for this policy: `count` for computing the number of the staffs and `sum` for summing their salaries. They are both recursive since the department can have any number of staffs. To express this policy, the *staff* element type modified with $*$ is changed into a protection type with the view component `()` and the embedded expression `()`, which means that all staffs should be hidden to the financial assistant. Moreover, another two protection types are added to construct the elements *number* and *totalsalary*, respectively. These two types do not protect any data in the original document, so their original components are both `()`.

```
fun count(x:⟨staff⟩[⟨name⟩[string], ⟨salary⟩[int]]*):int =
   if isempty(x) then 0 else 1+count(tail(x))
fun sum(x:⟨staff⟩[⟨name⟩[string], ⟨salary⟩[int]]*) : int =
   if isempty(x) then 0 else head(x)/child::salary/
   child::#/ϵ+sum(tail(x))
τ = ⟨department⟩[⟨name⟩[string], ⟨staffs⟩[
      ⟨staff⟩[⟨name⟩[string], ⟨salary⟩[int]]*↦ ()&(),
      () ↦ ⟨number⟩[int]&⟨number⟩[
        count(/child::department/child::staffs/child::
        staff/ϵ)],
      () ↦ ⟨totalsalary⟩[int]&⟨totalsalary⟩[
        sum(/child::department/child::staffs/child::
        staff/ϵ)]]]
```

Fig.5. Access policy for financial assistant.

## 3 Type System for CSchema

This section presents a type system to guarantee type correctness of CSchema. It intends to prevent the following type problems.

- The view component in a protection type is not the same as the type of the embedded expression.
- The embedded expression in a protection type has type errors.
- A path expression refers to wrong element names with respect to its context element.

In addition, this type system also generates security views for the successfully checked CSchema policies. A security view is just an ordinary regular expression type without including protection types. The type system is illustrated in two parts: the first is for CSchema correctness checking, and the second is for checking the types of embedded expressions. The first part depends on the second part.

### 3.1 Type Checking of CSchema

CSchema correctness checking, given in Fig.6, can prevent the first type problem. The judgment has the form $\Delta; \Theta; \Gamma \vdash_{\pi} \tau \Rightarrow \tau'$, which means under the environments $\Delta$, $\Theta$ and $\Gamma$, the type $\tau$ at the position $\pi$ is type correct and a corresponding security view $\tau'$ can be derived. In the judgment, the position $\pi$ is an absolute path recording the path from the root of the current policy to the $\tau$'s parent.

The environments are defined as follows:

$$\Delta ::= . \mid \Delta, t@\pi \quad \Theta ::= . \mid \Theta, t \mapsto \mu t.\tau \quad \Gamma ::= . \mid \Gamma, x : \tau$$

where $t@\pi$ means that the variable $t$ is bound at position $\pi$ in current policy; $t \mapsto \mu t.\tau$ maps the variable $t$ to its definition; $x : \tau$ means $x$ has the type $\tau$. In all cases, the dot . just plays a syntactic role and can be omitted when the environment is not empty. In the following, we introduce the rules for checking type variable and the protection type.

The rule for checking type variable $t$ has two premises. First, $t$ is asked to be bound at some position $\pi'$ in $\Delta$. Second, the path $\pi'$ should be a strict prefix of $\pi$ judged by the operator $\prec$ defined below. These premises check two syntactic requirements on CSchema policies. The first is that free type variables is now allowed in a correct policy, and the second is that any recursive variable (bound by $\mu$) must be guarded by at least an element name. For example, according to this rule, the recursive type $\mu t.t$ is not correct. This is to avoid nonterminated validation, which is illustrated more in Subsection 4.1. However, this rule also prevents the recursive types like $\mu t.\mathtt{int}, t$. This is not a really limitation for CSchema since this type can be represented as $\mathtt{int}, \mathtt{int}*$.

$$\epsilon \prec p, \text{if } p \neq \epsilon$$
$$/p_1 \prec /p_2, \text{if } p_1 \prec p_2$$
$$\mathtt{child} :: l/p_1 \prec \mathtt{child} :: l/p_2, \text{if } p_1 \prec p_2$$

In the rule for protection type $\tau_1 \mapsto \tau_2 \& e$, the type $\tau_2$ is checked under empty $\Delta$, $\Theta$ and $\Gamma$ because it cannot include any protection type. And, when checking $e$, $\mathtt{self}: \Theta\tau_1$ is preserved in $\Gamma$ because $e$ probably contains the subexpression $\mathtt{self::}l/p$ and consequently the

$$\frac{}{\Delta; \Theta; \Gamma \vdash_{\pi} b \Rightarrow b} \qquad \frac{t@\pi' \in \Delta \quad \pi' \prec \pi}{\Delta; \Theta; \Gamma \vdash_{\pi} t \Rightarrow t} \qquad \frac{\Delta; \Theta; \Gamma \vdash_{\pi/\mathtt{child}::l} \tau \Rightarrow \tau'}{\Delta; \Theta; \Gamma \vdash_{\pi} \langle l \rangle[\tau] \Rightarrow \langle l \rangle[\tau']}$$

$$\frac{\Delta; \Theta; \Gamma \vdash_{\pi} \tau_1 \Rightarrow \tau_1' \qquad \Delta; \Theta; \Gamma \vdash_{\pi} \tau_2 \Rightarrow \tau_2'}{\Delta; \Theta; \Gamma \vdash_{\pi} \tau_1, \tau_2 \Rightarrow \tau_1', \tau_2'} \qquad \frac{\Delta; \Theta; \Gamma \vdash_{\pi} \tau_1 \Rightarrow \tau_1' \quad \Delta; \Theta; \Gamma \vdash_{\pi} \tau_2 \Rightarrow \tau_2'}{\Delta; \Theta; \Gamma \vdash_{\pi} \tau_1 \mid \tau_2 \Rightarrow \tau_1' \mid \tau_2'}$$

$$\frac{\Delta; \Theta; \Gamma \vdash_{\pi} \tau \Rightarrow \tau'}{\Delta; \Theta; \Gamma \vdash_{\pi} \tau* \Rightarrow \tau'*} \qquad \frac{\Delta, t@\pi; \Theta, t \mapsto \mu t.\tau; \Gamma \vdash_{\pi} \tau \Rightarrow \tau'}{\Delta; \Theta; \Gamma \vdash_{\pi} \mu t.\tau \Rightarrow \mu t.\tau'}$$

$$\frac{\Delta; \Theta; \Gamma \vdash_{\pi} \tau_1 \Rightarrow \tau_1' \quad .;.;. \vdash_{\pi} \tau_2 \Rightarrow \tau_2 \quad \Gamma, \mathtt{self} : \Theta\tau_1 \vdash_{\pi} e : \tau_2 \quad \tau_2 :: \star}{\Delta; \Theta; \Gamma \vdash_{\pi} \tau_1 \mapsto \tau_2 \& e \Rightarrow \tau_2}$$

Fig.6. CSchema checking rules.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_{\pi} x : \tau} \qquad \frac{c \in \{\mathtt{true}, \mathtt{false}, n, s, ()\}}{\Gamma \vdash_{\pi} c : b} \qquad \frac{\Gamma \vdash_{\pi} e : \tau*}{\Gamma \vdash_{\pi} \mathtt{head}(e) : () \mid \tau} \qquad \frac{\Gamma \vdash_{\pi} e : \tau*}{\Gamma \vdash_{\pi} \mathtt{tail}(e) : \tau*}$$

$$\frac{\Gamma \vdash_{\pi} e : \tau*}{\Gamma \vdash_{\pi} \mathtt{isempty}(e) : \mathtt{bool}} \qquad \frac{\Gamma \vdash_{\pi} e : \mathtt{bool} \; \Gamma \vdash_{\pi} e_1 : \tau \; \Gamma \vdash_{\pi} e_2 : \tau}{\Gamma \vdash_{\pi} \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau} \qquad \frac{\Gamma \vdash_{\pi} e_1 : \mathtt{string} \; \Gamma \vdash_{\pi} e_2 : \mathtt{string}}{\Gamma \vdash_{\pi} e_1 =_{\mathrm{s}} e_2 : \mathtt{bool}}$$

$$\frac{\Gamma \vdash_{\pi} e_1 : \mathtt{int} \; \Gamma \vdash_{\pi} e_2 : \mathtt{int}}{\Gamma \vdash_{\pi} e_1 + e_2 : \mathtt{int}} \qquad \frac{\mathtt{fun}\ f(x : \tau_1) : \tau_2 = e' \in \Gamma \quad \tau_1 :: \star \quad \tau_2 :: \star \quad \Gamma, x : \tau_1 \vdash_{\pi} e' : \tau_2 \quad \Gamma \vdash_{\pi} e : \tau_1}{\Gamma \vdash_{\pi} f(e) : \tau_2}$$

$$\frac{\Gamma \vdash_{\pi} e : \tau}{\Gamma \vdash_{\pi} \langle l \rangle[e] : \langle l \rangle[\tau]} \qquad \frac{\Gamma \vdash_{\pi} e_1 : \tau_1 \quad \Gamma \vdash_{\pi} e_2 : \tau_2}{\Gamma \vdash_{\pi} e_1, e_2 : \tau_1, \tau_2} \qquad \frac{\Gamma \vdash_{\pi} e_1 : \tau_1 \quad \Gamma \vdash_{\pi} e_2 : \tau_2}{\Gamma \vdash_{\pi} e_1 \mid e_2 : \tau_1 \mid \tau_2}$$

$$\frac{\Gamma \vdash_{\pi} e : \tau' \quad \tau' \circ_{\mathtt{t}} p \Rightarrow \tau}{\Gamma \vdash_{\pi} e/p : \tau} (\mathtt{cond}(p, \tau)) \qquad \frac{\mathtt{doc} : \tau' \in \Gamma \quad \tau' \circ_{\mathtt{t}} p \Rightarrow \tau \quad /p \not\preceq \pi}{\Gamma \vdash_{\pi} /p : \tau} (\mathtt{cond}(p, \tau))$$

$$\frac{\mathtt{self} : \tau' \in \Gamma \quad \tau' \bullet_{\mathtt{t}} l \Rightarrow \tau'' \quad \tau'' \circ_{\mathtt{t}} p \Rightarrow \tau}{\Gamma \vdash_{\pi} \mathtt{self} :: l/p : \tau} (\mathtt{cond}(p, \tau)) \qquad \frac{\Gamma \vdash_{\pi} e : \tau_1 \quad [\tau_1]_{\mathrm{E}} \langle: [\tau_2]_{\mathrm{E}}}{\Gamma \vdash_{\pi} e : \tau_2}$$

Fig.7. Typing rules for expressions.

self's type is needed for typing $e$. $\Theta\tau_1$ is a type obtained by replacing all free variables in $\tau_1$ with their definitions in $\Theta$. Note that $\Theta\tau_1$ does not contain free type variables for a well-typed policy because all free variables in $\tau_1$ are required in the domain of $\Delta$ according to the typing rule for type variables, and $\Delta$ and $\Theta$ have the same domain according to the checking rule for recursive types.

Suppose there is a CSchema policy $\tau$, e.g., in Fig.4 or Fig.5. The judgment $. ; . ; \mathtt{doc} : \langle\mathtt{doc}\rangle[\tau] \vdash . \; \tau \Rightarrow \tau'$ is used to check it and generate the security view $\tau'$, in which the type of doc is kept for typing the absolute path expressions embedded in $\tau$. Finally, the following lemma says that a security view is an ordinary regular expression type.

**Lemma 1 (CSchema Checking).** *If* $. ; . ; \mathtt{doc} : \langle\mathtt{doc}\rangle[\tau] \vdash . \; \tau \Rightarrow \tau'$, *then* $\tau' :: \star$.

### 3.2 Typing Expressions

The typing rules for expressions are given in Fig.7, which can prohibit the second and the third type problems. The judgment is defined as $\Gamma \vdash_\pi e : \tau$, indicating that the expression $e$ has the type $\tau$ under the typing environment $\Gamma$. Some rules are illustrated in the following.

In the expression $e/p$, $e$ is the context element of the path $p$. To check the type of this expression, $e$ is first checked to get the context element type $\tau'$, under which $p$ is checked. Then, the operator $\circ_{\mathtt{t}}$ (formally defined later) is used to extract $\tau$ from $\tau'$ according to the path $p$. Finally, $\tau$ is the type of $e/p$ if $\mathtt{cond}(p, \tau)$ holds. The condition $\mathtt{cond}(p, \tau)$ is true if $p = \epsilon$, or if $p \neq \epsilon$ and $\tau \neq ()$. On the other hand, this condition fails if $p$ includes wrong element names or refers to the content of empty elements. These cases are regarded as static errors as discussed in [14].

The rule for the expression $\mathtt{self} :: l/p$ first determines the self's type $\tau'$ in $\Gamma$, which is preserved by the CSchema checking rule for protection type in last subsection. And then, the operator $\bullet_{\mathtt{t}}$ (formally defined later) is used to filter $\tau'$ with the label $l$ and get the new context element type $\tau''$ for the path $p$. The remaining checking is the same as that of the expression $e/p$.

The absolute path $/p$ has the context element doc. The type of doc is recorded in $\Gamma$ when beginning to check a CSchema policy. The typing rule is almost similar to that of $e/p$ except for the condition $/p \not\preceq \pi$, which means that $/p$ is not a prefix of $\pi$. This condition is to avoid circular reference among elements. The prefix relation $\preceq$ is a reflexive closure of $\prec$, that is, $p \preceq p$ or $p_1 \preceq p_2$ if $p_1 \prec p_2$.

The last rule is about type subsumption. The notation $[\tau]_{\mathrm{E}}$ denotes the type obtained by erasing all protection types in $\tau$. This operation is defined as follows:

$$
\begin{aligned}
[b]_{\mathrm{E}} &= b\\
[t]_{\mathrm{E}} &= t\\
[\langle l\rangle[\tau]]_{\mathrm{E}} &= \langle l\rangle[[\tau]_{\mathrm{E}}]\\
[\tau_1, \tau_2]_{\mathrm{E}} &= [\tau_1]_{\mathrm{E}}, [\tau_2]_{\mathrm{E}}\\
[\tau_1 \mid \tau_2]_{\mathrm{E}} &= [\tau_1]_{\mathrm{E}} \mid [\tau_2]_{\mathrm{E}}\\
[\tau_1 \mapsto \tau_2 \& e]_{\mathrm{E}} &= \tau_2\\
[\tau *]_{\mathrm{E}} &= [\tau]_{\mathrm{E}} *\\
[\mu\, t.\tau]_{\mathrm{E}} &= \mu\, t.[\tau]_{\mathrm{E}}.
\end{aligned}
$$

Erasure operation changes the protection type $\tau_1 \mapsto \tau_2 \& e$ into $\tau_2$. Since $\tau_2$ has the kind $\star$, $[\tau_1]_{\mathrm{E}}$ and $[\tau_2]_{\mathrm{E}}$ in the last rule are both ordinary regular expression types. $[\tau_1]_{\mathrm{E}} \; \langle: \; [\tau_2]_{\mathrm{E}}$ indicates that $[\tau_1]_{\mathrm{E}}$ is a subtype of $[\tau_2]_{\mathrm{E}}$. The subtype relation between two ordinary regular expression types can be determined by using the approach in [15]. The subsumption rule means that if expression $e$ has the type $\tau_1$ and $[\tau_1]_{\mathrm{E}} \; \langle: \; [\tau_2]_{\mathrm{E}}$, then $e$ also has the type $\tau_2$.

### 3.3 Position Typing Operators

The operation $\tau \circ_{\mathtt{t}} p$ extracts the constituent type of $\tau$ at position $p$ and $\tau \bullet_{\mathtt{t}} l$ is to filter type $\tau$ with label $l$. They are defined as follows:

$$
\begin{aligned}
\tau \circ_{\mathtt{t}} \epsilon &= \tau, \text{if } \tau \text{ is not a variable}\\
t \circ_{\mathtt{t}} p &= ()\\
b \circ_{\mathtt{t}} \mathtt{child} :: l/p &= ()\\
\langle l'\rangle[\tau] \circ_{\mathtt{t}} \mathtt{child} :: l/p &= (\tau \bullet_{\mathtt{t}} l) \circ_{\mathtt{t}} p\\
\tau_1, \tau_2 \circ_{\mathtt{t}} p &= \tau_1 \circ_{\mathtt{t}} p, \tau_2 \circ_{\mathtt{t}} p\\
\tau_1 \mid \tau_2 \circ_{\mathtt{t}} p &= \tau_1 \circ_{\mathtt{t}} p \mid \tau_2 \circ_{\mathtt{t}} p\\
\tau * \circ_{\mathtt{t}} p &= (\tau \circ_{\mathtt{t}} p) *\\
\mu\, t.\tau \circ_{\mathtt{t}} p &= \tau[t \mapsto \mu\, t.\tau] \circ_{\mathtt{t}} p\\
b \bullet_{\mathtt{t}} \# &= b\\
t \bullet_{\mathtt{t}} l &= ()\\
b \bullet_{\mathtt{t}} l &= ()\\
\langle l\rangle[\tau] \bullet_{\mathtt{t}} l &= \langle l\rangle[\tau]\\
\langle l\rangle[\tau] \bullet_{\mathtt{t}} l' &= (), \text{if } l \neq l'\\
\tau_1, \tau_2 \bullet_{\mathtt{t}} l &= \tau_1 \bullet_{\mathtt{t}} l, \tau_2 \bullet_{\mathtt{t}} l\\
\tau_1 \mid \tau_2 \bullet_{\mathtt{t}} l &= \tau_1 \bullet_{\mathtt{t}} l \mid \tau_2 \bullet l\\
\tau * \bullet_{\mathtt{t}} l &= (\tau \bullet_{\mathtt{t}} l) *\\
\tau_1 \mapsto \tau_2 \& e \bullet_{\mathtt{t}} l &= \tau_1 \bullet_{\mathtt{t}} l\\
\mu\, t.\tau \bullet_{\mathtt{t}} l &= \mu\, t.(\tau \bullet_{\mathtt{t}} l).
\end{aligned}
$$

There are several points worth some explanation.

First, $t \circ_{\mathtt{t}} p$ generates () definitely. If this case happens, it says that $t$ is free in the current policy because when extracting constituent type from the recursive type $\mu\, t.\tau$, it is unrolled, i.e., all free occurrences of $t$ in $\tau$ are substituted with $\mu\, t.\tau$.

Second, the operator $\circ_{\mathtt{t}}$ is not applied to protection type because this type is never used as a context element type. The reason is that if the context element is doc or self, their types are not protection types; on the other hand, if the context element type is generated by the operator $\bullet_{\mathtt{t}}$, they are not, either, according to the definition of $\bullet_{\mathtt{t}}$.

Third, a regular expression type is actually a sequence at its top level. For example, $\langle l\rangle[\tau]$ is a singleton sequence; $\tau_1, \tau_2$ is a sequence of two constituent types. Filtering a type with wrong element name will lead to an empty context element type.

Fourth, when filtering a protection type, $\tau_1$ is considered and $\tau_2$ is ignored. That is, when a path is across a protection type, the path is redirected to its original component.

### 3.4 Case Study 1: Policy Type-Checking

The access policy in Fig.5 will be checked in this section. For the convenience of presentation, the policy $\tau$ is broken into the following pieces:

$$
\begin{aligned}
\tau &= \langle\texttt{department}\rangle[\tau_1, \tau_2]\\
\tau_1 &= \langle\texttt{name}\rangle[\texttt{string}]\\
\tau_2 &= \langle\texttt{staffs}\rangle[\tau_3, \tau_4, \tau_5]\\
\tau_3 &= \langle\texttt{staff}\rangle[\langle\texttt{name}\rangle[\texttt{string}],\\
&\qquad\qquad \langle\texttt{salary}\rangle[\texttt{int}]]* \mapsto ()\&()\\
\tau_4 &= () \mapsto \langle\texttt{number}\rangle[\texttt{int}]\&e_1\\
\tau_5 &= () \mapsto \tau_6\&e_2\\
\tau_6 &= \langle\texttt{totalsalary}\rangle[\texttt{int}]\\
e_1 &= \langle\texttt{number}\rangle[\texttt{count}(p)]\\
e_2 &= \langle\texttt{totalsalary}\rangle[\texttt{sum}(p)]\\
p &= /\texttt{child} :: \texttt{department}/\texttt{child} :: \texttt{staffs}/p_0\\
p_0 &= \texttt{child} :: \texttt{staff}/\epsilon.
\end{aligned}
$$

The type correctness derivation of the policy $\tau$ is sketched in Fig.8, where $\Gamma = \texttt{doc} : \langle\texttt{doc}\rangle[\tau]$, $s_1 = \texttt{child} :: \texttt{department}$ and $s_2 = \texttt{child} :: \texttt{staffs}$. As shown in the derivation tree, after type checking, the security view only includes the accessible data for the assistant, and also, the embedded code in the policy is type correct.

## 4 Enforcement of Access Policy

Access policies in CSchema are enforced by the validation procedure in this section. A protected document is an XML file in external format, and after validated against a CSchema policy, it is changed into internal format and contains only the accessible data, which can then be accessed, for instance, by some XML query engines. This work uses the following syntax to represent XML documents in external format and internal format, respectively:

$$
\begin{aligned}
ev &::= \;() \mid s \mid \langle l\rangle[ev] \mid ev_1, ev_2\\
iv &::= \;() \mid s \mid n \mid \texttt{true} \mid \texttt{false} \mid \langle l\rangle[iv] \mid iv_1, iv_2.
\end{aligned}
$$

### 4.1 Validation

The validation procedure is defined in Fig.9. The judgment $\Phi; \Gamma \vdash ev \rhd \tau \Rightarrow iv$ indicates that validating the external value $ev$ against the type $\tau$ generates the internal value $iv$ under the environment $\Phi$ and $\Gamma$, where $\Gamma$ is the typing environment as before and $\Phi$ is defined as: $\Phi ::= . \mid \Phi, x = iv \mid \Phi, x = ev$, in which $x$ is bound to an internal value $iv$ or an external value $ev$. When validating a string $s$ against the type `bool` or `int`, the functions `s-to-b` and `s-to-i` convert $s$ into a Bool value or an integer, respectively. The rules for recursive type and protection type are explained in Fig.9.

When validating $ev$ against the recursive type $\mu t.\tau$, the result is that of validating $ev$ against the unrolled recursive type $\tau[t \mapsto \mu t.\tau]$. Recall that the correctness of CSchema requires that the free variable $t$ in $\tau$ is always

$$
\dfrac{\dfrac{\cdots}{.;.;\Gamma \vdash_{/s_1} \tau_1 \Rightarrow \tau_1} \quad \dfrac{\dfrac{\dfrac{\cdots}{.;.;\Gamma \vdash_{/s_1/s_2} \tau_3 \Rightarrow ()} \quad \dfrac{\cdots}{.;.;\Gamma \vdash_{/s_1/s_2} \tau_4 \Rightarrow \langle\texttt{number}\rangle[\texttt{int}]} \quad D}{.;.;\Gamma \vdash_{/s_1 s_2} \tau_3, \tau_4, \tau_5 \Rightarrow \langle\texttt{number}\rangle[\texttt{int}],\ \langle\texttt{totalsalary}\rangle[\texttt{int}]}}{.;.;\Gamma \vdash_{/s_1} \tau_2 \Rightarrow \langle\texttt{staffs}\rangle[\langle\texttt{number}\rangle[\texttt{int}], \langle\texttt{totalsalary}\rangle[\texttt{int}]]}}{\dfrac{.;.;\Gamma \vdash_{/s_1} \tau_1, \tau_2 \Rightarrow \tau_1, \langle\texttt{staffs}\rangle[\langle\texttt{number}\rangle[\texttt{int}], \langle\texttt{totalsalary}\rangle[\texttt{int}]]}{.;.;\Gamma \vdash .\tau \Rightarrow \langle\texttt{department}\rangle[\tau_1, \langle\texttt{staffs}\rangle[\langle\texttt{number}\rangle[\texttt{int}], \langle\texttt{totalsalary}\rangle[\texttt{int}]]]}}
$$

where, derivation $D$ is as follows:

$$
\dfrac{\dfrac{}{.;.;\Gamma \vdash_{/s_1/s_2} () \Rightarrow ()} \quad \dfrac{\cdots}{.;.;.\vdash_{/s_1 s_2} \tau_6 \Rightarrow \tau_6 \;\; \tau_6 :: \star} \quad \dfrac{\dfrac{\cdots}{\Gamma, \texttt{self} : () \vdash_{/s_1 s_2} \texttt{sum}(p) \Rightarrow \texttt{int}}}{\Gamma, \texttt{self} : () \vdash_{/s_1/s_2} e_2 \Rightarrow \tau_6}}{.;.;\Gamma \vdash_{/s_1/s_2} \tau_5 \Rightarrow \tau_6}
$$

Fig.8. Type checking of the example policy.

$$
\dfrac{}{\Phi; \Gamma \vdash () \rhd () \Rightarrow ()} \qquad \dfrac{}{\Phi; \Gamma \vdash s \rhd \texttt{bool} \Rightarrow \texttt{s-to-b}(s)} \qquad \dfrac{}{\Phi; \Gamma \vdash s \rhd \texttt{string} \Rightarrow s}
$$

$$
\dfrac{}{\Phi; \Gamma \vdash s \rhd \texttt{int} \Rightarrow \texttt{s-to-i}(s)} \qquad \dfrac{\Phi; \Gamma \vdash ev_1 \rhd \tau_1 \Rightarrow iv_1 \quad \Phi; \Gamma \vdash ev_2 \rhd \tau_2 \Rightarrow iv_2}{\Phi; \Gamma \vdash ev_1, ev_2 \rhd \tau_1, \tau_2 \Rightarrow iv_1, iv_2}
$$

$$
\dfrac{\Phi; \Gamma \vdash ev \rhd \tau_1 \Rightarrow iv}{\Phi; \Gamma \vdash ev \rhd \tau_1 \mid \tau_2 \Rightarrow iv} \qquad \dfrac{\Phi; \Gamma \vdash ev \rhd \tau_2 \Rightarrow iv}{\Phi; \Gamma \vdash ev \rhd \tau_1 \mid \tau_2 \Rightarrow iv} \qquad \dfrac{\Phi; \Gamma \vdash ev \rhd () \mid \tau, \tau* \Rightarrow iv}{\Phi; \Gamma \vdash ev \rhd \tau* \Rightarrow iv}
$$

$$
\dfrac{\Phi; \Gamma \vdash ev \rhd \tau[t \mapsto \mu t.\tau] \Rightarrow iv}{\Phi; \Gamma \vdash ev \rhd \mu t.\tau \Rightarrow iv} \qquad \dfrac{\Phi; \Gamma \vdash ev \rhd \tau \Rightarrow iv}{\Phi; \Gamma \vdash \langle l\rangle[ev] \rhd \langle l\rangle[\tau] \Rightarrow \langle l\rangle[iv]} \qquad \dfrac{\Phi, \texttt{self} = ev; \Gamma, \texttt{self} : \tau_1 \vdash e \Downarrow iv}{\Phi; \Gamma \vdash ev \rhd \tau_1 \mapsto \tau_2\&e \Rightarrow iv}
$$

Fig.9. Validating rules.

guarded by at least one element name. Hence, validating against the unrolled recursive type does not directly incur this rule again, which can prevent against nonterminated validation caused by the continuous attempt of trying this rule.

The rule of validating $ev$ against a protection type performs access control actions. This rule does not validate $ev$ and instead the internal value at the $ev$'s position is obtained by evaluating the embedded expression $e$ in this protection type. That is, this rule not only hides the sensitive data but also creates the accessible data. The evaluation of $e$ depends on the dynamic semantics of expressions. Before evaluating an expression, the `self` with value $ev$ and type $\tau_1$ are preserved in $\Phi$ and $\Gamma$, respectively. Thus, when $e$ refers to `self`, its value and type can be found in these environments.

Suppose there is a protected document $ev$ and a CSchema $\tau$. The access policy $\tau$ should be enforced under the following initial environments: $\Phi$ contains $\texttt{doc} = \langle\texttt{doc}\rangle[ev]$, and $\Gamma$ is $\texttt{doc} : \langle\texttt{doc}\rangle[\tau]$. It should be mentioned that if $\tau$ does not include any protection type, then this validation procedure will not evaluate any expression, and consequently cost nothing for access control purpose; if the access policy just hides some sensitive data, then this validation procedure is more efficient than ordinary ones since it eliminates the need to validate the sensitive data. For an extreme example, if the whole document should be hidden, it returns an empty value immediately without validating the document at all.

### 4.2 Dynamic Semantics

The dynamic semantics of expressions is presented in Fig.10. The judgment has the form $\Phi;\Gamma \vdash e \Downarrow iv$. That is, under the environments $\Phi$ and $\Gamma$, evaluating the expression $e$ generates an internal value $iv$. Some representative rules are illustrated below.

When applied to a sequence value, the `head` expression returns its first item and the `tail` expression returns its tail. The $\texttt{isempty}(e)$ expression judges whether $e$ evaluates to an empty value. Evaluation of the function application $f(e)$ needs to evaluate the function body $e'$ under a new environment obtained by adding the binding $x = iv$ into $\Phi$, where $iv$ is the evaluation result of the argument expression $e$.

In the expression $e/p$, the value of $e$ is the context element of the path $p$. To evaluate this expression, $e$ is evaluated first to the value $iv'$; and then, the value at the position $p$ of $iv'$ is extracted by the operator $\circ_{\mathtt{v}}$ and used as the value of the expression $e/p$. The definitions of $\circ_{\mathtt{v}}$ and $\bullet_{\mathtt{v}}$ are given at the end of this subsection but without too much explanation since they have similar behaviors as the corresponding type operators $\circ_{\mathtt{t}}$ and $\bullet_{\mathtt{t}}$.

For the path expression $/p$ (or $\texttt{self::}l/p$), its context element `doc` (or `self`) is still represented in the external format, so the value at the position $p$ is also in the external format. After this value is extracted by using the operator $\circ_{\mathtt{v}}$, it is converted into the internal format by validating against its type. Hence, the values specified by these path expressions are actually the values in the released document. If a path expression wants to refer to a sensitive value, it must penetrate into the original component of the protection type that protects this value. In addition, according to the typing rule for $/p$, the expression in a protection type is prohibited from referring to its parent type, so evaluating $/p$ with some context element will not cause evaluate this expression again.

$$
\begin{aligned}
iv \circ_{\mathtt{v}} \epsilon &= iv\\
c \circ_{\mathtt{v}} \texttt{child} :: l/p &= ()\\
\langle l'\rangle[iv] \circ_{\mathtt{v}} \texttt{child} :: l/p &= (iv \bullet_{\mathtt{v}} l) \circ_{\mathtt{v}} p\\
iv_1, iv_2 \circ_{\mathtt{v}} p &= iv_1 \circ_{\mathtt{v}} p, iv_2 \circ_{\mathtt{v}} p\\
iv \bullet_{\mathtt{v}} \# &= iv\\
c \bullet_{\mathtt{v}} l &= ()\\
\langle l\rangle[iv] \bullet_{\mathtt{v}} l &= \langle l\rangle[iv]\\
\langle l\rangle[iv] \bullet_{\mathtt{v}} l' &= (), \text{if } l \neq l'\\
iv_1, iv_2 \bullet_{\mathtt{v}} l &= iv_1 \bullet_{\mathtt{v}} \mathtt{l}, iv_2 \bullet_{\mathtt{v}} l
\end{aligned}
$$

where $c \in \{(), s, n, \texttt{true}, \texttt{false}\}$.

$$
\frac{x = iv \in \Phi}{\Phi;\Gamma \vdash x \Downarrow iv}
\quad
\frac{c \in \{(), s, n, \texttt{false}, \texttt{true}\}}{\Phi;\Gamma \vdash c \Downarrow c}
\quad
\frac{\Phi;\Gamma \vdash e_1 \Downarrow n_1 \quad \Phi;\Gamma \vdash e_2 \Downarrow n_2 \quad n = n_1 + n_2}{\Phi;\Gamma \vdash e_1 + e_2 \Downarrow n}
$$

$$
\frac{\Phi;\Gamma \vdash e_1 \Downarrow s_1 \quad \Phi;\Gamma \vdash e_2 \Downarrow s_2 \quad iv = (s_1 \text{ equals } s_2)}{\Phi;\Gamma \vdash e_1 =_{\mathtt{s}} e_2 \Downarrow iv}
\quad
\frac{\Phi;\Gamma \vdash e \Downarrow iv_1, iv_2}{\Phi;\Gamma \vdash \texttt{head}(e) \Downarrow iv_1}
\quad
\frac{\Phi;\Gamma \vdash e \Downarrow iv_1, iv_2}{\Phi;\Gamma \vdash \texttt{tail}(e) \Downarrow iv_2}
$$

$$
\frac{\Phi;\Gamma \vdash e \Downarrow ()}{\Phi;\Gamma \vdash \texttt{isempty}(e) \Downarrow \texttt{true}}
\quad
\frac{\Phi;\Gamma \vdash e \Downarrow iv \quad iv \neq ()}{\Phi;\Gamma \vdash \texttt{isempty}(e) \Downarrow \texttt{false}}
\quad
\frac{\Phi;\Gamma \vdash e \Downarrow iv}{\Phi;\Gamma \vdash \langle l\rangle[e] \Downarrow \langle l\rangle[iv]}
$$

$$
\frac{\Phi;\Gamma \vdash e \Downarrow \texttt{true} \quad \Phi;\Gamma \vdash e_1 \Downarrow iv_1}{\Phi;\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Downarrow iv_1}
\quad
\frac{\Phi;\Gamma \vdash e \Downarrow \texttt{false} \quad \Phi;\Gamma \vdash e_2 \Downarrow iv_2}{\Phi;\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Downarrow iv_2}
\quad
\frac{\Phi;\Gamma \vdash e_1 \Downarrow iv_1 \quad \Phi;\Gamma \vdash e_2 \Downarrow iv_2}{\Phi;\Gamma \vdash e_1, e_2 \Downarrow iv_1, iv_2}
$$

$$
\frac{\Phi(\texttt{self}) \bullet_{\mathtt{v}} l \Rightarrow ev \quad ev \circ_{\mathtt{v}} p \Rightarrow ev' \quad \Gamma(\texttt{self}) \bullet_{\mathtt{t}} l \Rightarrow \tau \quad \tau \circ_{\mathtt{t}} p \Rightarrow \tau' \quad \Phi;\Gamma \vdash ev' \rhd \tau' \Rightarrow iv}{\Phi;\Gamma \vdash \texttt{self} :: l/p \Downarrow iv}
$$

$$
\frac{\Phi;\Gamma \vdash e \Downarrow iv' \quad iv' \circ_{\mathtt{v}} p \Rightarrow iv}{\Phi;\Gamma \vdash e/p \Downarrow iv}
\quad
\frac{\Phi(\texttt{doc}) \circ_{\mathtt{v}} p \Rightarrow ev \quad \Gamma(\texttt{doc}) \circ_{\mathtt{t}} p \Rightarrow \tau \quad \Phi;\Gamma \vdash ev \rhd \tau \Rightarrow iv}{\Phi;\Gamma \vdash /p \Downarrow iv}
\quad
\frac{\texttt{fun}\ f(x:\tau_1):\tau_2 = e' \in \Gamma \quad \Phi;\Gamma \vdash e \Downarrow iv \quad \Phi, x = iv;\Gamma \vdash e' \Downarrow iv'}{\Phi;\Gamma \vdash f(e) \Downarrow iv'}
$$

Fig.10. Dynamic semantics.

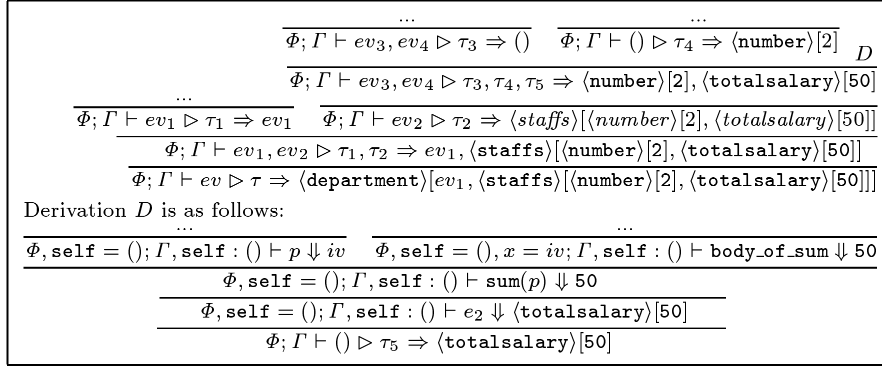


Fig.11. Enforcement of the example policy.

### 4.3 Case Study 2: Policy Enforcement

In this subsection, the access policy in Fig.5 will be enforced on the protected file in Fig.1(a), which is represented as follows in the external format:

$$ev = \langle\texttt{departments}\rangle[ev_1, ev_2]$$
$$ev_1 = \langle\texttt{name}\rangle[\texttt{"Market"}]$$
$$ev_2 = \langle\texttt{staffs}\rangle[ev_3, ev_4]$$
$$ev_3 = \langle\texttt{staff}\rangle[\langle\texttt{name}\rangle[\texttt{"Tom"}], \langle\texttt{salary}\rangle[\texttt{"30"}]]$$
$$ev_4 = \langle\texttt{staff}\rangle[\langle\texttt{name}\rangle[\texttt{"Peter"}], \langle\texttt{salary}\rangle[\texttt{"20"}]].$$

This file will be validated against the policy in Fig.5 to produce the file in Fig.1(b) in the internal format. For convenience, the policy notations in Subsection 3.4 are still used here. The validation procedure is sketched in Fig.11, where $\Phi$ includes $\texttt{doc} = \langle\texttt{doc}\rangle[ev]$, $\Gamma$ contains $\texttt{doc} : \langle\texttt{doc}\rangle[\tau]$, $D$ is the derivation for validating empty value () against $\tau_5$, and $iv$ is the following internal value of the path expression $p$:

$$\langle\texttt{staff}\rangle[\langle\texttt{name}\rangle[\texttt{"Tom"}], \langle\texttt{salary}\rangle[30]],$$
$$\langle\texttt{staff}\rangle[\langle\texttt{name}\rangle[\texttt{"Peter"}], \langle\texttt{salary}\rangle[20]].$$

As the example shows, after policy enforcement, the assistant only gets the accessible data with respect to the access policy.

### 4.4 Property of CSchema

To state the property of CSchema, the semantics of types with the kind $\star$ is needed. The semantics function $[\![_]\!]$ maps a type to a set of internal values, defined as follows:

$$[\![()]\!] = \{()\}$$
$$[\![\texttt{int}]\!] = \{n\}$$
$$[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\}$$
$$[\![\texttt{string}]\!] = \{s\}$$
$$[\![\langle l\rangle[\tau]]\!] = \{\langle l\rangle[iv] \mid iv \in [\![\tau]\!]\}$$
$$[\![\tau_1, \tau_2]\!] = \{iv_1, iv_2 \mid iv_1 \in [\![\tau_1]\!], iv_2 \in [\![\tau_2]\!]\}$$
$$[\![\tau_1 \mid \tau_2]\!] = [\![\tau_1]\!] \cup [\![\tau_2]\!]$$
$$[\![\tau*]\!] = \{()\} \cup [\![\tau, \tau*]\!]$$
$$[\![\mu t.\tau]\!] = [\![\tau[t \mapsto \mu t.\tau]]\!].$$

Then, the following theorem says that when validating a protected document against a CSchema policy, a released document is generated and it must match the security view derived from this policy.

**Theorem 1 (Access Policy Enforcement).** *Suppose that $\tau$ is a CSchema policy and $ev$ a protected document. If $.\,;.\,;\Gamma \vdash .\ \tau \Rightarrow \tau'$ and $\Phi; \Gamma \vdash ev \rhd \tau \Rightarrow iv$, then $iv \in [\![\tau']\!]$, where $\Phi$ contains the binding $\texttt{doc} = \langle\texttt{doc}\rangle[ev]$ and $\Gamma$ includes the type information $\texttt{doc} : \langle\texttt{doc}\rangle[\tau]$.*

The proof proceeds by induction over the CSchema checking rules and validation rules for each type. For brevity, only the case for the protection type is sketched. By induction hypothesis, it is known that $e$ has type $\tau_2$ and $iv'$ is the evaluation result of $e$. And $\tau_2$ has the kind $\star$, so $iv' \in [\![\tau_2]\!]$.

## 5 Discussion

This section will discuss other possible applications of CSchema, compare it with other XML processing languages, and explain how to use CSchema with these existing languages.

### 5.1 Other Possible Applications

Though CSchema is motivated to describe downgrading access policies for XML access control, actually it can also be used to maintain the computation dependency found in XML documents[16]. For example, in an XML file for a book, its table of contents depends on the titles of its chapters. Fig.12 gives a CSchema policy

```
fun mktoc(x: ⟨chap⟩[⟨title⟩[string],⟨p⟩[string]*]*):
    ⟨title⟩[string]*
          = if isempty(x) then () else head(x)/child::
          title/ϵ,mktoc(tail(x))
τ = ⟨book⟩[⟨toc⟩[() ↦ ⟨title⟩[string]*&mktoc(/child::
            book/child::chap/ϵ)],
            ⟨chap⟩[⟨title⟩[string],⟨p⟩[string]*]*]
```

Fig.12. CSchema policy for book.

for such a case, where a book consists of a table of contents and a sequence of chapters. And, the table of contents includes a sequence of chapter titles, which are computed from the sequence of `chap` elements by applying the function `mktoc` to the argument `/child::book/child::chap/`$\epsilon$.

### 5.2 Comparison with Other Approaches

Some XML processing languages like XQuery[17] and CDuce[11], or general programming languages enhanced with XML processing ability like Java and C$\omega$[18], can transform the original document in Fig.1(a) to the released one in Fig.1(b). That is, these languages can be used to write policies. However, writing policies with these languages are very clumsy since they do not allow the policy writers to focus on the sensitive data.

For example, suppose a leaf element in a big document is supposed to be sensitive and should be hidden. Then the policy if written in CDuce has to use a pattern designed by considering the whole document to separate the sensitive leaf element from other insensitive elements and then use the insensitive ones to construct a new document; the policy if written in XQuery needs to use many XPath expressions to extract the insensitive elements to construct the released document since they are both functional languages. For imperative languages like Java or C$\omega$, the policy needs not construct a new released document, however it still needs to navigate to the sensitive element and remove it from the document.

In CSchema, the policy writers just need to change the type of the protected leaf element into a protection type with the empty view component `()` and the empty expression `()`, and then all work is done. Moreover, the policy in CSchema is enforced when validating this XML document, and thus the sensitive element is removed by simply ignoring it during validation and thus such enforcement does not incur any overhead.

### 5.3 Using CSchema with Other Languages

Though its advantage in enforcing XML access control, CSchema is not designed to compete with the existing XML processing languages, and rather CSchema can be used together with them as the access control module. The general idea is to find the validation modules in existing languages and then conservatively extend them to support protection type for CSchema without changing the validation method to other standard types, such as element types or sequence types.

Two examples are given to explain how to incorporate CSchema into some existing languages. The first example is about the Galax XQuey engine[19] (an implementation of XQuery). As shown in its architecture[20], there is an explicit stage *Schema Validation* to validate the input XML document against an XML Schema. In order to use CSchema in the Galax engine, we just need to extend the *Schema Validation* module by supporting protection type. The second example is about the CDuce[11]. This language does not has an explicit processing stage to valid an input XML document. Its validation is implemented by a language construct `match`, which matches an untyped input XML document in the external form against a type, and generates the XML data of this type in the internal form if this document really has this type. In order to use CSchema, the construct `match` is needed to extend to support protection type.

## 6 Related Work

The current approaches of XML access control are only able to release the accessible elements or hide the inaccessible elements that are already existing in the protected documents. For example, the approach in [1] is a DTD-based mechanism, and the elements specified with the structure of DTD, are only associated with the access mode Y (accessible) or N (inaccessible); the approaches in [5–7] specify the elements only with the permissions of read granting or read denying; the elements in [2, 3] are specified only with positive authorization + or negative authorization −; the browsing privileges in [4] include view, navigate, and browse_all, and they describe different conditions for reading the existing attributes or elements in documents.

Active XML (AXML)[21] allows code to be embedded in XML documents, not in schema like the approach in this work. Moreover, the code in an AXML is just web service calling to integrate other data into the current document, and cannot describe downgrading operations for access control purpose. However, the embedded operations in CSchmea can be used to exploit web services if the appropriate functions are used. The work[16] is more general than AXML, but it also embeds code in XML documents, not in schema. These approaches have the shortcoming that if two XML documents have the same schema (or same structure), they cannot let these documents share the embedded code.

Language-based security[22] is a very active research area. Language technologies have been applied to such areas as certified code[23], stack inspection[24], etc. But none has used these technologies to protect the content of XML documents. The existing downgrading policy languages[10,25] concern the information flow among code, while CSchema moves the focus to the tree-structured data.

## 7 Conclusion

This paper motivates the downgrading problem for XML access control and proposes CSchema to address this problem. By downgrading policies in CSchema, such access control case can be expressed, where the original sensitive element keeps inaccessible, but the elements computed from them become insensitive and can be released to data requesters. In addition, CSchema

can also be used to maintain the computation dependencies among one XML document.

CSchema extends the ordinary schema languages by *protection type*, which is embedded by an expression to describe the downgrading operation. A type system for CSchema is also presented, and it not only checks the type correctness of CSchema, but also generates security views. Among the existing approaches, only method in [1] can provide security view. Access policies represented in CSchema are enforced by a conservative extension of ordinary validation procedure and it can deal with the validation of XML files against ordinary schema as well.

CSchema does not conflict with other XML processing technologies. For example, it can be used together with XQuery by combining its validation procedure into query engines.

**Acknowledgment** Thanks to the support from Prof. Masato Takeichi and Prof. Zhen-Jiang Hu, and discussions with other PSD project members in the University of Tokyo. We are also grateful to the anonymous reviewers for their comments.

## References

[1] Wenfei Fan, Chee-Yong Chan, Minos Garofalakis. Secure XML querying with security views. In *Proc. the 2004 ACM Int. Conf. Management of Data*, Paris, France, 2004, pp.587–598.

[2] Damiani E, di Vimercati S, Paraboschi S *et al.* Securing XML documents. In *Proc. Int. Conf. Extending Database Technology,* Konstanz, Germany, 20000, *LNCS 1777*, pp.121–135.

[3] Damiani E, di Vimercati S, Paraboschi S *et al.* A fine-grained access control system for XML documents. *ACM Trans. Information and System Security*, 2002, 5(2): 2: 169–202.

[4] Bertino E, Ferrari E. Secure and selective dissemination of XML documents. *ACM Trans. Information and System Security,* 2002, 5(3): 290–331.

[5] Gabilon A, Bruno E. Regulating access to XML documents. In *Working Conf. Database and Application Security*, Ontario, Canada, 2001, pp.299–314.

[6] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, Satoshi Hada. XML access control using static analysis. In *Proc. 10th ACM Conf. Computer and Communications Security*, Washington DC, USA, 2003, pp.73–84.

[7] Hada S, Kudo M. XML access control language: Provisional authorization for XML documents. 2000, http://www.trl.ibm.com/projects/xml/xacl.

[8] Godik S, Moses T. eXtensible access control markup 2 language (XACML) Version 1.0. 2003, http://www.oasis-open.org/specs/index.php.

[9] Irini Fundulaki, Maarten Marx. Specifying access control policies for XML documents with XPath. In *Proc. the 9th ACM Symp. Access Control Models and Technologies*, New York, USA, 2004, pp.61–69.

[10] Stephen Chong, Andrew C Myers. Security policies for downgrading. In *Proc. the 11th ACM Conf. Computer and Communications Security*, Washington DC, USA, 2004, pp.198–209.

[11] Veronique Benzaken, Giuseppe Castagna, Alain Frisch. CDuce: An XML-centric general-purpose language. In *Proc. the 8th Int. Conf. Functional Programming*, Uppsala, Sweden, 2003, pp.51–63.

[12] Jerome Simeon, Philip Wadler. The essence of XML. In *Proc. the 30th ACM Symp. Principles of Programming Languages*, New Orleans, Louisiana, USA, 2003, pp.1–13.

[13] Hosoya H, Pierce B C. XDuce: A typed XML processing language. *ACM Trans. Internet Technology*, 2003, 3(2): 117–148.

[14] Dario Colazzo, Giorgio Ghelli, Paolo Manghi, Carlo Sartiani. Types for path correctness of XML queries. In *Proc. the 9th ACM Int. Conf. Functional Programming*, Snowbird, USA, 2004, pp.126–137.

[15] Haruo Hosoya, Jerome Vouillon, Benjamin C Pierce. Regular expression types for XML. In *Proc. the 5th ACM Int. Conf. Functional Programming*, Montreal, Canada, 2000, pp.11–22.

[16] Dongxi Liu, Zhenjiang Hu, Masato Takeichi. An environment for maintaining computation dependency in XML documents. In *Proc. the 2005 ACM Symp. Document Engineering*, Bristol, UK, 2005, pp.42–51.

[17] W3C Recommendation. XML Query (XQuery). 2005, http://www.w3.org/XML/Query.

[18] Bierman G, Meijer E, Schulte W. The essence of data access in Comega. In *European Conf. Object-Oriented Programming, LNCS 3586*, Glasgow, UK, 2005, pp.287–311.

[19] The Galax Team. Galax: An Implementation of XQuery. http://www.galaxquery.org.

[20] Mary Fernandez, Jerome Simeon. Build your own XQuery processor. In *EDBT Summer School*, Sardinia, Italy, 2004.

[21] Serge Abiteboul, Omar Benjelloun, Tova Milo. Positive active XML. In *Proc. the 23rd ACM Symp. Principles of Database Systems*, Paris, France, 2004, pp.35–45.

[22] Schneider F B, Morrisett G, Harperi R. A language-based approach to security. *Informatics: 10 Years Back, 10 Years Ahead, LNCS 2000*, Springer-Verlag, 2000, pp.86–101.

[23] George C Necula. Proof-carrying code. In *Proc. the 24th ACM Symp. Principles of Programming Languages*, Paris, France, 1997, pp.106–119.

[24] Cedric Fournet, Andrew D Gordon. Stack inspection: Theory and variants. In *Proc. the 29th ACM Symp. Principles of Programming Languages*, Portland, Oregon, USA, 2002, pp.307–318.

[25] Peng Li, Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. Principles of Programming Languages*, Long Beach, California, 2005, pp.158–170.

**Dong-Xi Liu** received his B.E. and M.E. degrees from Taiyuan Univ. Technology in 1996 and 1999, and his Ph.D. degree from Shanghai JiaoTong Univ. in 2003, respectively. He has been a researcher in the University of Tokyo since 2004, and before that he worked at National University of Singapore for one year as a research fellow. His current research interests include language-based security, software verification based on advanced type systems, and bidirectional transformation language design for XML processing.