SeLINQ: Tracking Information across Application-Database Boundaries

Daniel Schoepe Daniel Hedin Andrei Sabelfeld

Chalmers University of Technology, Gothenburg, Sweden

Abstract

The root cause for *confidentiality* and *integrity* attacks against computing systems is insecure *information flow*. The complexity of modern systems poses a major challenge to secure *end-to-end* information flow, ensuring that the insecurity of a single component does not render the entire system insecure. While information flow in a variety of languages and settings has been thoroughly studied in isolation, the problem of tracking information across component boundaries has been largely out of reach of the work so far. This is unsatisfactory because tracking information across component boundaries is necessary for end-to-end security.

This paper proposes a framework for uniform tracking of information flow through both the application and the underlying database. Key enabler of the uniform treatment is recent work by Cheney et al., which studies database manipulation via an embedded language-integrated query language (with Microsoft's LINQ on the backend). Because both the host language and the embedded query languages are functional F#-like languages, we are able to leverage information-flow enforcement for functional languages to obtain information-flow control for databases "for free", synergize it with information-flow control for applications and thus guarantee security across application-database boundaries. We develop the formal results in the form of a security type system that includes a treatment of algebraic data types and pattern matching, and establish its soundness. On the practical side, we implement the framework and demonstrate its usefulness in a case study with a realistic movie rental database.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—information-flow controls

Keywords end-to-end security, information flow, static analysis, language-integrated queries

1. Introduction

Increasingly, we trust interconnected software on desktops, laptops, tablets, and smart phones to manipulate a wide range of sensitive information such as medical, commercial, and location information. This trust can be justified only if the software is designed, constructed, monitored, and audited to be robust and secure.

Securing heterogeneous systems *Heterogeneity* is a major roadblock in the path of software security. Modern computing systems are built with a large number of components, often run on different platforms and written in multiple programming languages.

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-2873-9 /14/09. . . $15.00.
http://dx.doi.org/10.1145/2628136.2628151

It is not surprising that systems often break at component boundaries. The OWASP Top 10 project identifies ten most critical web application security risks [2]. The top of the list is dominated by attacks across component boundaries: injection attacks (with SQL injection as prime example) are number 1 on the list; cross-site scripting attacks are number 3. In both, untrusted data bypasses inter-component filtering, which leads executing malicious commands (commonly in SQL or JavaScript) to compromise *confidentiality* and *integrity*.

In the face of complexity and heterogeneity of today's systems, it is vital to ensure *end-to-end security* [45], overarching component boundaries.

Information-flow control The root cause for confidentiality and integrity attacks against computing systems is insecure *information flow*. For confidentiality, this implies a possibility of leaking information from sensitive sources to attacker-observable sinks. For integrity, this implies a possibility of data from untrusted sources to compromise data on trusted sinks.

Enforcing secure information flow is more involved than enforcing safety properties like tracking units of measure [33] or taint tracking [47]. This is due to the fact that there are two different types of information flows. The first type of flow, the *explicit* flows, originates from the explicit propagation of values, via, e.g., parameter passing. Tracking this kind of flows is similar to tracking units of measure or taint tracking. The second type of flows, the *implicit* [25] flows, corresponds to flows via the control flow. Consider

```
l = if (h) then true else false
```

Depending on the value of h, either the *then* branch or the *else* branch of the conditional is chosen to be evaluated to give the final result. In the above program, this has the effect of leaking the Boolean value of h into l, constituting an implicit flow from h to l. A different machinery is needed to track this kind of flows, which distinguishes enforcement of secure information flow from enforcement of safety properties [51].

A large, extensively surveyed [13, 29, 30, 41], body of work has studied information-flow control. However, with a few recent exceptions (discussed in Section 6), the problem of information flow for different components has largely been explored in isolation. This is unsatisfactory because tracking information across component boundaries is necessary for end-to-end security.

Motivated by the above, this paper focuses on information-flow control for systems with database components.

Database integration Programs commonly access databases via libraries that connect and interact with the database. If we take SQL as an example, querying is typically done by constructing a query string that is passed to the database as illustrated below.

```
let query  = "SELECT Name FROM People";
let result = SqlCommand(query, db).execute();
```

The problem with this approach is that the queries are constructed at runtime without any guarantees on the query. In general it is hard to verify that the constructed queries are meaningful let alone decide information flow properties for the queries. The cre-

ated string could be an invalid query or even the result of an SQL injection. Further, the returned information is by necessity encoded in a generic way, which makes it both inefficient and error prone to work with. Instead, it is attractive to integrate database query mechanism into the language as facilitated, e.g, by Google's Web Toolkit [4], Ruby on Rails [11], and Microsoft's LINQ [5].

In functional setting, an elegant approach to provide language-integrated query is to use meta-programming based on quotations and antiquotations. This is the approach taken by Cheney et al. [16]. The goal is to provide access to SQL databases in F# (with Microsoft's LINQ on the backend). F# provides quotation via `<@ @>`, which creates a typed representation of a given F# expression e. Assuming that e has type t, then `<@` e `@>` is a value of type **Expr**$\langle t \rangle$. Antiquotes `(%  )` provide a way to splice in typed quoted values into other quoted expressions. This approach capitalizes on the flexible meta-programming capabilities of F# [50]. With this framework we can express the above query in F# in the following way.

```
let query =
<@ for p in (% db).People do
     yield p.Name
@>
let result = run query
```

From the type of the spliced in database, `db`, the type system of F# is able to determine the type of `query` to **Expr**$\langle$**list string**$\rangle$. In turn `query` is given to `run` which, when run, executes the query resulting in a list of **string**s. The typing of the program is compile time, whereas the creation and execution of the actual query is runtime. At runtime the quoted expression is parsed by the F# runtime and the typed result is passed to `run` for normalization and evaluation. This produces and performs the actual SQL query. Note how antiquotation is used to splice in the database allowing the construction of multiple queries using the same database connection.

Contributions This paper puts homogeneous meta-programming to work to develop information-flow type systems for heterogeneous systems. In particular we present an information-flow type system for a subset of F# with database queries. The presented development is an instance of a general method that allows for the reuse of existing type systems to create information flow type systems that seamlessly spans language boundaries. Thus, the method is not limited to database queries.

Because both the host language and the embedded query languages are F#-like subsets, we are able to leverage information-flow enforcement for functional languages to obtain information-flow control for databases "for free". The simplicity of the resulting type system and the relatively small modifications needed is evidence for the success of the approach.

In a nutshell, the paper contains the following main contribution:

(i) We leverage homogeneous meta-programming to provide information-flow security for a subset of F# including database access via the essence of query processing in Microsoft LINQ, as it is expressed in F#.

In addition, the paper contains further contributions:

(ii) We develop the formal results in the form of a security type system and show that it enforces the security condition of *noninterference* [28] (Section 2).

(iii) We develop an analysis to treat algebraic data types and pattern matching, establish its soundness, and implement it as a part of our prototype (Section 4).

(iv) We present an implementation of the type checker and a translator from our language to executable F# code (Section 3).

(v) We demonstrate the usefulness of our framework by a case study with a realistic movie rental database (Section 5).

The full soundness proof and the code of the framework and case study are available online[1].

2. Framework

This section presents a simple functional language with support for product types, records, lists, quoted expressions and antiquotations, the security type system, and shows that the type system enforces information-flow security with respect to a small-step semantics.

Recall that the fundamental idea is that, since the information-flow of the database interaction is fully described in the quoted language, the type systems is able to enforce information-flow security for the database interactions for free.

2.1 Language

The language is based on the one used by Cheney et al. [16] with the addition of security levels to the type system.

Figure 1 shows the syntax of security levels, types, and terms. We write $\overline{x}$ to denote a sequence of entities x. For example, $\overline{f : t}$ is a shorthand for a sequence $f_1 : t_1, f_2 : t_2, \ldots, f_n : t_n$ of typings of record fields.

$$\ell ::= \mathtt{L} \mid \mathtt{H}$$

$$b ::= \textbf{int}^{\ell} \mid \textbf{string}^{\ell} \mid \textbf{bool}^{\ell}$$

$$t ::= b \mid t \to t \mid t * t \mid \{\overline{f : t}\} \mid (t\ \textbf{list})^{\ell} \mid \textbf{Expr}\langle t \rangle$$

$$T ::= (\{\overline{f : b}\})\ \textbf{list}^{\ell}$$

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x : t$$

$$\begin{aligned} e ::= {} & c \mid x \mid op(\overline{e}) \mid \textbf{lift}\ e \mid \textbf{fun}(x) \to e \mid \textbf{rec}\ f(x) \to e \mid (e, e) \\ \mid {} & \textbf{fst}\ e \mid \textbf{snd}\ e \mid \{\overline{f = e}\} \mid e.f \mid \textbf{yield}\ e \mid [] \\ \mid {} & e\ @\ e \mid \textbf{for}\ x\ \textbf{in}\ e\ \textbf{do}\ e \mid \textbf{exists}\ e \mid \textbf{if}\ e\ \textbf{then}\ e \mid \textbf{run}\ e \\ \mid {} & \texttt{<@}\ e\ \texttt{@>} \mid \texttt{(\%}\ e\ \texttt{)} \mid \textbf{database}(x) \end{aligned}$$

Figure 1. Syntax of language and types

We remark on some of the interesting constructs: c denotes built-in constants, such as integers and booleans. op denotes built-in operators, such as addition and logical connectives. **lift** e lifts an expression of type t to type **Expr**$\langle t \rangle$. **for** x **in** e_1 **do** e_2 is used to express list comprehensions where x is bound successively to elements in e_1 when evaluating e_2. The results of evaluating e_2 for each element are then concatenated. **run** e denotes running a quoted expression e. This involves generating an SQL query based on the quoted term. e_1 @ e_2 denotes concatenation of e_1 and e_2. Section 2.2 provides further details. **exists** e evaluates to **true** if and only if the expression e does not evaluate to the empty list. This can be used to check if the result of a query is empty. Similarly, **if** e_1 **then** e_2 evaluates to e_2 if e_1 evaluates to a non-empty list and to [] otherwise. **yield** e denotes a singleton list consisting of expression e. `<@` e `@>` denotes a quoted expression e. The language allows only closed quoted terms, since this simplifies the semantics of the language and is still able to express all the desired concepts. Quoted functions can be expressed by abstracting in the quoted term as opposed to abstracting on the level of the host language. `(%` e `)` denotes antiquotation of the expression e, and allows splicing of quoted expressions into quoted expressions in a type-safe way.

Security type language The security type language is defined by annotating a standard type language for a functional fragment with

[1] `http://www.cse.chalmers.se/~schoepe/selinq/`

quotations with security levels ℓ. Without loss of generality the security levels are taken from the two-element security lattice consisting of a level `L` for non-confidential information and a level `H` for confidential information. Information-flow integrity policies can be expressed dually [14]. The types are split into base types (b), which can occur as types of columns in tables (T), and general types (t) which include function types, lists, and quoted expressions.

As is common, we consider a database to be a collection of tables. Each table consists of at least one named column, each of which is equipped with a fixed security level annotated type. The security levels on types for database columns express which columns contain confidential data and which columns do not.

To express security policies for databases, each database is given a type signature. Such a type signature describes tables as lists of records. Each record field corresponds to a column in the sense that the field name matches the name of the column in the database. A column is specified as confidential or public by using a suitable type for the corresponding field in the record. The ordering of elements in a list used to represent table contents is irrelevant.

To illustrate the addition of security levels to the type system in the case of databases, consider an example, adapted from Cheney et al. [16], involving a database of people and couples, `PeopleDB`. In this scenario, we assume that the names of people are confidential, while the age is not, which leads to the following type for `PeopleDB`.

```
PeopleDB :
  { People :
    { Id : intˆL; Name : stringˆH; Age : intˆL } listˆL
  ; Couples :
    { Person1 : intˆL ; Person2 : intˆL } listˆL
  }
```

Now consider the situation where we want to query the database for couples where one partner is more than 10 years older than the other partner. This can be done by iterating once over all couples in the database and then iterating twice over all people in the database. For each couple and pair of persons, one then checks if they are part of the couple that is being considered and checks if the age difference is higher than 10. If that is the case, the name of the first partner along with the age difference is returned as part of the result, which is a list of records consisting of a name and the age difference.

```
let db = <@ database "PeopleDB" @>

type ResultType = {name : stringˆH ; diff : intˆL}

let differences : Expr < ResultType list ˆ L > =
  <@ for c in (% db).Couples do
     for p1 in (% db).People do
     for p2 in (% db).People do
     if (c.Person1 = p1.Id) &&
        (c.Person2 = p2.Id) &&
        (abs (p1.Age - p2.Age) > 10) then
     yield ({ name = p1.Name
            ; diff = p1.Age - p2.Age })
  @>

let main = run differences
```

As can be seen in the above program the information-flow policy for this program is specified by giving a type annotation to the quoted expression that generates the query, i.e., a type annotation for `differences`. In particular, the name components of the result are typed confidential, while the age differences are public. This matches the policy specified for the database contents, in which the names of people are confidential while their ages are not. The type system ensures that the result type of `differences` is in fact compatible with the policy specified for the database. Changing the security annotation of the `name` field from secret to public as follow results in a type error.

```
// No longer well-typed:
type ResultType = {name : stringˆL ; diff : intˆL}
```

2.2 Operational Semantics

We denote evaluation of an expression e using database data in Ω to another expression e' by $e \longrightarrow_\Omega e'$. Ω is a function that maps database names to the actual content of the database it refers to, and δ is a mapping that maps operators to their corresponding semantics. Σ maps constants and databases to their respective types.

We assume that Ω is consistent with the typing for databases given in Σ: for each database $\Omega(db)$ is assumed to be a value of type $\Sigma(db)$.

The evaluation rules in Figures 2, 3, 4, and 5 follow [16]. Let $\longrightarrow_\Omega^*$ be the reflexive-transitive closure of $\longrightarrow_\Omega$. Evaluation and normalization of the quoted language is denoted by $eval_\Omega(norm(e))$. This evaluation entails generating database queries that can be executed by actual database servers. In particular, higher-order features such as nested records or function applications need to be evaluated to obtain computations that can be expressed in SQL. Figure 6 shows the syntax. The semantics is call-by-value with left-to-right evaluation of terms. This is formalized using evaluation contexts $\mathcal{E}$. Quotation contexts $\mathcal{Q}$ are used to ensure that there are no antiquotations left of the hole.

We denote substitution of free occurrences of a variable x in expression e with another expression e' by $e[x \mapsto e']$.

$$
\begin{aligned}
V ::= {} & c \mid \mathbf{fun}(x) \to e \mid \mathbf{rec}\ f(x) \to e \mid (V, V) \mid \{\overline{f = V}\} \\
& \mid [] \mid \mathbf{yield}\ V \mathbin{@} \ldots \mathbin{@} \mathbf{yield}\ V \mid \texttt{<@}\ Q\ \texttt{@>} \\
\\
Q ::= {} & c \mid op(\overline{Q}) \mid \mathbf{lift}\ Q \mid x \mid \mathbf{fun}(x) \to Q \mid Q\ Q \mid (Q, Q) \\
& \mid \{\overline{f = Q}\} \mid Q.f \mid \mathbf{yield}\ Q \mid [] \mid Q \mathbin{@} Q \mid \mathbf{for}\ x\ \mathbf{in}\ Q\ \mathbf{do}\ Q \\
& \mid \mathbf{exists}\ Q \mid \mathbf{if}\ Q\ \mathbf{then}\ Q \mid \mathbf{database}(db) \\
\\
\mathcal{E} ::= {} & [] \mid op(\overline{V}, \mathcal{E}, \overline{M}) \mid \mathbf{lift}\ \mathcal{E} \mid \mathcal{E}\ e \mid V\ \mathcal{E} \mid (\mathcal{E}, e) \mid (V, \mathcal{E}) \\
& \mid \{\overline{f = V}, f' = \mathcal{E}, \overline{f = e}\} \mid \mathcal{E}.f \mid \mathbf{yield}\ \mathcal{E} \mid \mathcal{E} \mathbin{@} e \mid V \mathbin{@} \mathcal{E} \\
& \mid \mathbf{for}\ x\ \mathbf{in}\ \mathcal{E}\ \mathbf{do}\ e \mid \mathbf{exists}\ \mathcal{E} \mid \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ e \mid \mathbf{run}\ \mathcal{E} \\
& \mid \texttt{<@}\ \mathcal{Q}[\texttt{(\%}\ \mathcal{E}\ \texttt{)}]\ \texttt{@>} \\
\\
\mathcal{Q} ::= {} & [] \mid op(\overline{Q}, \mathcal{Q}, \overline{e}) \mid \mathbf{fun}(x) \to \mathcal{Q} \mid \mathbf{lift}\ \mathcal{Q} \mid \mathcal{Q}\ e \mid V\ \mathcal{Q} \\
& \mid (\mathcal{Q}, e) \mid (Q, \mathcal{Q}) \mid \{\overline{f = Q}, f' = \mathcal{Q}, \overline{f = e}\} \mid \mathcal{Q}.f \\
& \mid \mathbf{yield}\ \mathcal{Q} \mid \mathcal{Q} \mathbin{@} e \mid V \mathbin{@} \mathcal{Q} \mid \mathbf{for}\ x\ \mathbf{in}\ \mathcal{Q}\ \mathbf{do}\ e \mid \mathbf{for}\ x\ \mathbf{in}\ Q\ \mathbf{do}\ \mathcal{Q} \\
& \mid \mathbf{exists}\ \mathcal{Q} \mid \mathbf{if}\ \mathcal{Q}\ \mathbf{then}\ e \mid \mathbf{if}\ Q\ \mathbf{then}\ \mathcal{Q} \mid \mathbf{run}\ \mathcal{Q}
\end{aligned}
$$

Figure 2. Values and evaluation contexts

2.3 Security Condition

The goal of the type system is to enforce a notion of noninterference for functional language. Noninterference formalizes computational independence between secrets and non-secrets, guaranteeing that no information about the former can be inferred from the latter. More precisely, this is expressed as the preservation of an equivalence relation under pairwise execution; given two inputs that are equal in the components that are visible to an attacker, evaluation should result in two output values that also coincide in the components that can be observed by the attacker.

To that end this section introduces a notion of low-equivalence denoted by $\sim$ that demands that parts of values with types that are annotated with `L` are equal, while placing no demands on the secret counterparts. More formally, we introduce a family of equivalence relations on values parametrized by types.

$$
\begin{aligned}
op(\overline{V}) &\longrightarrow \delta(op, \overline{V}) \\
(\textbf{fun}(x) \to N)\ V &\longrightarrow N[x \mapsto V] \\
(\textbf{rec}\ f(x) \to N)\ V &\longrightarrow M[f \mapsto \textbf{rec}\ f(x) \to N, x \mapsto V] \\
\textbf{fst}\ (V_1, V_2) &\longrightarrow V_1 \\
\textbf{snd}\ (V_1, V_2) &\longrightarrow V_2 \\
\{\overline{f = V}\}.f_i &\longrightarrow V_i \\
\textbf{if true then}\ M &\longrightarrow M \\
\textbf{if false then}\ M &\longrightarrow [] \\
\textbf{for}\ x\ \textbf{in yield}\ V\ \textbf{do}\ M &\longrightarrow M[x \mapsto V] \\
\textbf{for}\ x\ \textbf{in}\ []\ \textbf{do}\ N &\longrightarrow [] \\
\textbf{for}\ x\ \textbf{in}\ L \mathbin{@} M\ \textbf{do}\ N &\longrightarrow (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ N) \mathbin{@} (\textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N) \\
\textbf{exists}\ [] &\longrightarrow \textbf{false} \\
\textbf{exists}\ [\overline{V}] &\longrightarrow \textbf{true}, \qquad |\overline{V}| > 0 \\
\textbf{run}\ Q &\longrightarrow eval(norm(Q)) \\
\textbf{lift}\ c &\longrightarrow \texttt{<@}\ c\ \texttt{@>} \\
\texttt{<@}\ \mathcal{Q}[\texttt{(\%}\ \texttt{<@}\ Q\ \texttt{@>}\ \texttt{)}]\ \texttt{@>} &\longrightarrow \texttt{<@}\ \mathcal{Q}[Q]\ \texttt{@>}
\end{aligned}
$$

$$
\frac{M \longrightarrow N}{\mathcal{E}[M] \longrightarrow \mathcal{E}[N]}
$$

Figure 3. Evaluation rules for host language

$$
\begin{aligned}
(\textbf{fun}(x) \to R)\ Q &\rightsquigarrow R[x \mapsto Q] \\
\{\overline{f = Q}\}.f_i &\rightsquigarrow Q_i \\
\textbf{for}\ x\ \textbf{in yield}\ Q\ \textbf{do}\ R &\rightsquigarrow R[x \mapsto Q] \\
\textbf{for}\ y\ \textbf{in}\ (\textbf{for}\ x\ \textbf{in}\ P\ \textbf{do}\ Q)\ \textbf{do}\ R &\rightsquigarrow \textbf{for}\ x\ \textbf{in}\ P\ \textbf{do}\ (\textbf{for}\ y\ \textbf{in}\ Q\ \textbf{do}\ R) \\
\textbf{for}\ x\ \textbf{in}\ (\textbf{if}\ P\ \textbf{then}\ Q)\ \textbf{do}\ R &\rightsquigarrow \textbf{if}\ P\ \textbf{then}\ (\textbf{for}\ x\ \textbf{in}\ Q\ \textbf{do}\ R) \\
\textbf{for}\ x\ \textbf{in}\ []\ \textbf{do}\ N &\rightsquigarrow [] \\
\textbf{for}\ x\ \textbf{in}\ (P \mathbin{@} Q)\ \textbf{do}\ R &\rightsquigarrow \\
&\quad (\textbf{for}\ x\ \textbf{in}\ P\ \textbf{do}\ R) \mathbin{@} (\textbf{for}\ x\ \textbf{in}\ Q\ \textbf{do}\ R) \\
\textbf{if true then}\ Q &\rightsquigarrow Q \\
\textbf{if false then}\ Q &\rightsquigarrow []
\end{aligned}
$$

Figure 4. Symbolic reduction phase

$$
\begin{aligned}
\textbf{for}\ x\ \textbf{in}\ P\ \textbf{do}\ (Q \mathbin{@} R) &\hookrightarrow \\
&\quad (\textbf{for}\ x\ \textbf{in}\ P\ \textbf{do}\ Q) \mathbin{@} (\textbf{for}\ x\ \textbf{in}\ P\ \textbf{do}\ R) \\
\textbf{for}\ x\ \textbf{in}\ P\ \textbf{do}\ [] &\hookrightarrow [] \\
\textbf{if}\ P\ \textbf{then}\ (Q \mathbin{@} R) &\hookrightarrow (\textbf{if}\ P\ \textbf{then}\ Q) \mathbin{@} (\textbf{if}\ P\ \textbf{then}\ R) \\
\textbf{if}\ P\ \textbf{then}\ [] &\hookrightarrow [] \\
\textbf{if}\ P\ \textbf{then}\ (\textbf{if}\ Q\ \textbf{then}\ R) &\hookrightarrow \textbf{if}\ P\ \texttt{\&\&}\ Q\ \textbf{then}\ R \\
\textbf{if}\ P\ \textbf{then}\ (\textbf{for}\ x\ \textbf{in}\ Q\ \textbf{do}\ R) &\hookrightarrow \textbf{for}\ x\ \textbf{in}\ Q\ \textbf{do}\ (\textbf{if}\ P\ \textbf{then}\ R)
\end{aligned}
$$

Figure 5. Ad-hoc reduction phase

Definition 1 ($\sim_t$)**.** *The family of equivalence relations $\sim_t$ is defined inductively by the rules in figure 7.*

$$
\begin{aligned}
S &::= [] \mid X \mid X \mathbin{@} X \\
X &::= \textbf{database}(db) \mid \textbf{yield}\ Y \mid \textbf{if}\ Z\ \textbf{then yield}\ Y \\
&\quad \mid \textbf{for}\ x\ \textbf{in database}(db).f\ \textbf{do}\ X \\
Y &::= x \mid \{\overline{f = Z}\} \\
Z &::= c \mid x.f \mid op(\overline{X}) \mid \textbf{exists}\ S
\end{aligned}
$$

Figure 6. Normalized terms

$$
\frac{\ell = \mathtt{L} \Rightarrow i = i'}{i \sim_{\textbf{int}^\ell} i'} \qquad \frac{\ell = \mathtt{L} \Rightarrow s = s'}{s \sim_{\textbf{string}^\ell} s'} \qquad \frac{\ell = \mathtt{L} \Rightarrow b = b'}{b \sim_{\textbf{bool}^\ell} b'}
$$

$$
\frac{\begin{array}{c}\forall v_1, v_2, v_1', v_2', \Omega_1, \Omega_2.(\Omega_1 \sim_\Sigma \Omega_2 \wedge v_1 \sim_t v_2 \wedge \\ e_1[x \mapsto v_1] \longrightarrow^*_{\Omega_1} v_1' \wedge e_2[x \mapsto v_2] \longrightarrow^*_{\Omega_2} v_2') \Rightarrow \\ v_1' \sim_{t'} v_2'\end{array}}{\textbf{fun}(x) \to e_1 \sim_{t \to t'} \textbf{fun}(x) \to e_2}
$$

$$
\frac{\begin{array}{c}\forall v_1, v_2, v_1', v_2', \Omega_1, \Omega_2. \\ \Omega_1 \sim_\Sigma \Omega_2 \wedge v_1 \sim_t v_2 \wedge \\ e_1[f \mapsto \textbf{rec}\ f(x) \to e_1, x \mapsto v_1] \longrightarrow^*_{\Omega_1} v_1' \wedge \\ e_2[f \mapsto \textbf{rec}\ f(x) \to e_2, x \mapsto v_2] \longrightarrow^*_{\Omega_2} v_2' \Rightarrow \\ v_1' \sim_{t'} v_2'\end{array}}{\textbf{rec}\ f(x) \to e_1 \sim_{t \to t'} \textbf{rec}\ f(x) \to e_2}
$$

$$
\frac{v_1 \sim_{t_1} v_1' \qquad v_2 \sim_{t_2} v_2'}{(v_1, v_2) \sim_{t_1 * t_2} (v_1', v_2')} \qquad \frac{\overline{v \sim_t w}}{\{\overline{f = v}\} \sim_{\{\overline{f:t}\}} \{\overline{f = w}\}}
$$

$$
\frac{\ell = \mathtt{L} \Rightarrow (|[\overline{v}]| = |[\overline{w}]| \wedge \overline{v \sim_t w})}{[\overline{v}] \sim_{(t\ \textbf{list})^\ell} [\overline{w}]}
$$

$$
\frac{\begin{array}{c}\forall \Omega_1, \Omega_2.\Omega_1 \sim \Omega_2 \Rightarrow \\ eval_{\Omega_1}(norm(e_1)) \sim_t eval_{\Omega_2}(norm(e_2))\end{array}}{e_1 \sim_{\textbf{Expr}\langle t \rangle} e_2}
$$

Figure 7. Introduction rules for $\sim_t$

When the type is evident from the context, we omit the subscript on $\sim$. Moreover, we also write $\sim$ for sequences of values.

To present the relations in a more concise manner, we combine the cases for different security levels using implication in the premises; e.g. equality on base types is only required if the security level is L.

Base types are compared using ordinary equality if the values are considered public. In the case of function types and quoted expressions, $\sim_t$ corresponds to noninterference for the bodies of the functions.

Records are related by $\sim$ if they contain the same fields, and each field's contents are also related by $\sim$. Two lists are required to have the same length if the list type is annotated with L, but their contents may differ based on the element type.

To illustrate this, consider two lists of integers $l_1 =$ **yield** $1 \mathbin{@} []$ and $l_2 =$ **yield** $2 \mathbin{@} []$. If the lists are typed with the type $t = (\textbf{int}^{\mathtt{H}}\ \textbf{list})^{\mathtt{L}}$, the length of the list is considered public, while the contents are confidential. If in contrast the type is $t' = (\textbf{int}^{\mathtt{L}}\ \textbf{list})^{\mathtt{L}}$, neither the contents nor the length of the list is confidential. Hence $l_1 \sim_t l_2$ holds while $l_1 \sim_{t'} l_2$ does not.

For simplicity, $\sim_t$ is stated from the point of view of an observer on level L. $\sim_t$ can be generalized for an arbitrary lattice by

parametrizing it with the level of the observer. Instead of checking if the level annotation on the type is equal to H, one then checks if it is higher than the level of the observer.

Let Ω be a mapping from database names to database contents. We define low-equivalence for database mappings structurally in the following way.

Definition 2 ($\sim_\Sigma$)**.** *$\Omega_1 \sim_\Sigma \Omega_2$ holds if and only if for all databases db it holds that $\Omega_1(db) \sim_{\Sigma(db)} \Omega_2(db)$*

With this we are ready to define the top-level notion of security, based on *noninterference* [28]. Since the family of low-equivalence relations is parametrized by types the definition is done with respect to the initial database type and the final result type.

Definition 3 ($NI(e_1, e_2)_{\Sigma,t}$)**.** *Two expression e_1 and e_2 are non-interfering with respect to the database type Σ and the exit type t if for all Ω_1, Ω_2, v_1 and v_2 such that $\Omega_1 \sim_\Sigma \Omega_2$, and $e_i \longrightarrow^*_{\Omega_i} v_i$ for $i \in \{1, 2\}$ it holds that*

$$v_1 \sim_t v_2$$

In particular for any given closed expression e, $NI(e, e)_{\Sigma,t}$ should be read as e is secure with respect to the security policy expressed by Σ and t, i.e., no secret parts of the database as defined by Σ is able to influence the public parts of the returned value as defined by t.

As common [1, 38, 48] in this setting, noninterference is *termination-insensitive* [41, 52] in the sense that leaks via the observation of (non)termination are ignored.

2.4 Type System

Figure 8 presents the typing rules for the host language. Typing judgments are of the form $\Gamma \vdash e : t$ where Γ is a typing context mapping variables to types, e is an expression, and t is a type. It denotes that expression e has type t in context Γ. $\ell \sqcup \ell'$ denotes the join of levels ℓ and ℓ', i.e., $\ell \sqcup \ell' = \mathtt{H}$ iff $\mathtt{H} \in \{\ell, \ell'\}$, and $\ell \sqcup \ell' = \mathtt{L}$ otherwise.

Figure 9 presents the typing rules for the quoted language. Typing judgments in the quoted language have the form $\Gamma; \Delta \vdash e : t$, where Γ is the typing context for the host language and Δ is the typing context for the quoted language.

Most types contain a level annotation ℓ that denotes whether or not the "structure" of the value is confidential. In the case of base types such **int** or **string**, this means that their values are confidential or not. In the case of $(t\ \mathbf{list})^\ell$, the level ℓ indicates whether or not the length of the list is confidential. If $\ell = \mathtt{H}$, the entire list value is considered a secret, but if the $\ell = \mathtt{L}$, the length of the list may be disclosed to a public observer. However, the elements of the list may or may not be confidential depending on the level of the elements given by the type t.

Record types, functions, and quoted expression types do not carry an explicit level annotation, since their security level is contained in sub-components of the type.

In the case of records, it suffices to annotate the type of each field, since the structure of a record can not be modified dynamically. The confidentiality of a function is contained in the level annotation on the result type. The intuition is that, in the absence of side effects, the only way for a function to disclose information is via its result. For types for quoted expressions, i.e., types of the form $\mathbf{Expr}\langle t\rangle$, the level annotation is already contained in t.

We assume that types for operators, constants, and databases are given by the mapping Σ. Moreover, we also assume that each query only uses a single database.

The typing rules for expressions in the host language and expressions in the quoted language are nearly identical with a few exceptions:

- Recursion is only allowed in the host language.
- Quotations are only allowed in the host language.
- Expressions of the form **database**(x) are only allowed in the quoted language.
- Antiquotations are only allowed in the quoted language.

$$\textsc{Const}\ \frac{\Sigma(c) = t}{\Gamma \vdash c : t^\ell} \qquad \textsc{Var}\ \frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \textsc{Lift}\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{lift}\ e : \mathbf{Expr}\langle t\rangle}$$

$$\textsc{Fun}\ \frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \mathbf{fun}(x) \to e : (t \to t')} \qquad \textsc{Rec}\ \frac{\Gamma, x : t, f : t \to t' \vdash e : t'}{\Gamma \vdash \mathbf{rec}\ f(x) \to e : t \to t'}$$

$$\textsc{Apply}\ \frac{\Gamma \vdash e_1 : t \to t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : t'} \qquad \textsc{Op}\ \frac{\Sigma(op) = \overline{t} \to t \qquad \overline{\Gamma \vdash e : t^\ell}}{\Gamma \vdash op(\overline{e}) : t^{\bigsqcup \ell_i}}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 * t_2} \qquad \textsc{Fst}\ \frac{\Gamma \vdash e : t_1 * t_2}{\Gamma \vdash \mathbf{fst}\ e : t_1}$$

$$\textsc{Snd}\ \frac{\Gamma \vdash e : t_1 * t_2}{\Gamma \vdash \mathbf{snd}\ e : t_2} \qquad \textsc{Record}\ \frac{\overline{\Gamma \vdash M : t}}{\Gamma \vdash \{\overline{f = M}\} : \{\overline{f : t}\}}$$

$$\textsc{Project}\ \frac{\Gamma \vdash L : \{\overline{f : t}\}}{\Gamma \vdash L.f_i : t_i} \qquad \textsc{Yield}\ \frac{\Gamma \vdash M : t}{\Gamma \vdash \mathbf{yield}\ M : (t\ \mathbf{list})^\ell} \qquad \textsc{Nil}\ \frac{}{\Gamma \vdash [] : (t\ \mathbf{list})^\ell}$$

$$\textsc{Union}\ \frac{\Gamma \vdash M : (t\ \mathbf{list})^\ell \qquad \Gamma \vdash N : (t\ \mathbf{list})^{\ell'}}{\Gamma \vdash N\ @\ M : (t\ \mathbf{list})^{\ell \sqcup \ell'}}$$

$$\textsc{For}\ \frac{\Gamma \vdash M : (t\ \mathbf{list})^\ell \qquad \Gamma, x : t \vdash N : (t'\ \mathbf{list})^{\ell'}}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N : (t'\ \mathbf{list})^{\ell \sqcup \ell'}}$$

$$\textsc{Exists}\ \frac{\Gamma \vdash M : (t\ \mathbf{list})^\ell}{\Gamma \vdash \mathbf{exists}\ M : \mathbf{bool}^\ell} \qquad \textsc{If}\ \frac{\Gamma \vdash L : \mathbf{bool}^\ell \qquad \Gamma \vdash M : (t\ \mathbf{list})^{\ell'}}{\Gamma \vdash \mathbf{if}\ L\ \mathbf{then}\ M : (t\ \mathbf{list})^{\ell \sqcup \ell'}}$$

$$\textsc{Run}\ \frac{\Gamma \vdash M : \mathbf{Expr}\langle t\rangle}{\Gamma \vdash \mathbf{run}\ M : t} \qquad \textsc{Quote}\ \frac{\Gamma; \cdot \vdash M : t}{\Gamma \vdash \texttt{<@}\ M\ \texttt{@>} : \mathbf{Expr}\langle t\rangle}$$

$$\textsc{Sub}\ \frac{\ell \leq \ell' \qquad \Gamma \vdash M : t^\ell}{\Gamma \vdash M : t^{\ell'}}$$

Figure 8. Type system for host language

When lists are constructed using **yield** and [] they can be assigned an arbitrary level. Expressions of the form e_1 @ e_2 reveal information about the structure of both lists and hence their security levels are combined in the result type. Similarly, **exists** only reveals information about the structure of the list, but nothing about the contents. Therefore, the security level of list contents is discarded and only the security level of the list itself is present in the result type.

$$\textsc{ConstQ}\ \frac{\Sigma(c) = t}{\Gamma; \Delta \vdash c : t^{\ell}} \qquad \textsc{FunQ}\ \frac{\Gamma; \Delta, x : t \vdash e : t'}{\Gamma; \Delta \vdash \mathbf{fun}(x) \rightarrow e : t \rightarrow t'}$$

$$\textsc{VarQ}\ \frac{x : t \in \Delta}{\Gamma; \Delta \vdash x : t} \qquad \textsc{ApplyQ}\ \frac{\Gamma; \Delta \vdash e_1 : t \rightarrow t' \qquad \Gamma; \Delta \vdash e_2 : t}{\Gamma; \Delta \vdash e_1\ e_2 : t'}$$

$$\textsc{OpQ}\ \frac{\Sigma(op) = \overline{t} \rightarrow t \qquad \overline{\Gamma; \Delta \vdash M : t^{\ell}}}{\Gamma; \Delta \vdash op(\overline{M}) : t^{\bigsqcup \ell_i}}$$

$$\textsc{PairQ}\ \frac{\Gamma; \Delta \vdash e_1 : t_1 \qquad \Gamma; \Delta \vdash e_2 : t_2}{\Gamma; \Delta \vdash (e_1, e_2) : t_1 * t_2} \qquad \textsc{FstQ}\ \frac{\Gamma; \Delta \vdash e : t_1 * t_2}{\Gamma; \Delta \vdash \mathbf{fst}\ e : t_1}$$

$$\textsc{SndQ}\ \frac{\Gamma; \Delta \vdash e : t_1 * t_2}{\Gamma; \Delta \vdash \mathbf{snd}\ e : t_2} \qquad \textsc{RecordQ}\ \frac{\overline{\Gamma; \Delta \vdash M : t}}{\Gamma; \Delta \vdash \{\overline{f = M}\} : \{\overline{f : t}\}}$$

$$\textsc{ProjectQ}\ \frac{\Gamma; \Delta \vdash L : \{\overline{f : t}\}}{\Gamma; \Delta \vdash L.f_i : t_i} \qquad \textsc{YieldQ}\ \frac{\Gamma; \Delta \vdash M : t}{\Gamma; \Delta \vdash \mathbf{yield}\ M : (t\ \mathbf{list})^{\ell}}$$

$$\textsc{NilQ}\ \frac{}{\Gamma; \Delta \vdash [] : (t\ \mathbf{list})^{\ell}} \qquad \textsc{ExistsQ}\ \frac{\Gamma; \Delta \vdash M : (t\ \mathbf{list})^{\ell}}{\Gamma; \Delta \vdash \mathbf{exists}\ M : \mathbf{bool}^{\ell}}$$

$$\textsc{IfQ}\ \frac{\Gamma; \Delta \vdash L : \mathbf{bool}^{\ell} \qquad \Gamma; \Delta \vdash M : (t\ \mathbf{list})^{\ell'}}{\Gamma; \Delta \vdash \mathbf{if}\ L\ \mathbf{then}\ M : (t\ \mathbf{list})^{\ell \sqcup \ell'}}$$

$$\textsc{UnionQ}\ \frac{\Gamma; \Delta \vdash M : (t\ \mathbf{list})^{\ell} \qquad \Gamma; \Delta \vdash N : (t\ \mathbf{list})^{\ell'}}{\Gamma; \Delta \vdash N\ @\ M : (t\ \mathbf{list})^{\ell \sqcup \ell'}}$$

$$\textsc{ForQ}\ \frac{\Gamma; \Delta \vdash M : (t\ \mathbf{list})^{\ell} \qquad \Gamma; \Delta, x : t \vdash N : (t'\ \mathbf{list})^{\ell'}}{\Gamma; \Delta \vdash \mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N : (t'\ \mathbf{list})^{\ell \sqcup \ell'}}$$

$$\textsc{SubQ}\ \frac{\ell \leq \ell' \qquad \Gamma; \Delta \vdash M : t^{\ell}}{\Gamma; \Delta \vdash M : t^{\ell'}} \qquad \textsc{DatabaseQ}\ \frac{\Sigma(db) = \{\overline{f : t}\}}{\Gamma; \Delta \vdash \mathbf{database}(db) : \{\overline{f : t}\}}$$

$$\textsc{Antiquote}\ \frac{\Gamma \vdash e : \mathbf{Expr}\langle t \rangle}{\Gamma; \Delta \vdash (\%\ e\ ) : t}$$

Figure 9. Typing rules for quoted language

Note that the rule QUOTE ensures that its arguments are typed in an empty context for quoted expressions. This expresses that only closed quoted terms are allowed in this language. Running a quoted expression e of type $\mathbf{Expr}\langle t \rangle$ using **run** e results in a an expression of type t (rule RUN).

Expressions of the for $\mathbf{database}(db)$ get their type from the mapping Σ. The rule ANTIQUOTE allows to reference entities defined in the host language from within a quoted expression. The argument of an antiquotation must itself be a quoted expression.

The rules SUB and SUBQ allows raising the security level of an expression. $\ell \leq \ell'$ holds if and only if $\ell = \mathtt{L} \vee \ell = \ell' = \mathtt{H}$.

To illustrate the type system further, we explain the typing rule FOR rule in greater detail. Recall that **for** expressions are used to denote list comprehensions. The typing rule assigns the resulting list the join of the security level of both sub-expressions. The following two examples demonstrate why this is required.

Consider the following program that uses a **for** expression to leak the structure of the lists xs and ys. We assume xs to have type $(t\ \mathbf{list})^{\ell}$ for some type t and level ℓ, whereas ys has type $(t'\ \mathbf{list})^{\ell'}$.

```
for x in xs do ys
```

Since the resulting lists for each element of xs will be concatenated, the resulting list will have length $|xs| \times |ys|$ where $|a|$ denotes the length of a. If either xs or ys contains only one element, the length of the other list is revealed through the result. To account for this information flow, the resulting list will be typed with level $\ell \sqcup \ell'$.

2.5 Soundness Result

As explained above, the soundness result is stated in terms of noninterference, i.e., as the preservation of a low-equivalence relation under pairwise execution. If we start out in any two low-equivalent environments then the result of running a well-typed program will be low-equivalent with respect to the type of the program.

Assuming that the typing of the execution environment corresponds to the capabilities of the attacker, noninterference guarantees that all information readable by the attacker is independent of confidential information. To make the connection between the database policy Σ and the type system explicit we write $\Sigma \vdash e : t$ even though Σ was kept implicit in the type rules in Figures 8 and 9.

Theorem 1 (Typing soundness)**.** *If* $\Sigma \vdash e : t$*, then* $NI(e, e)_{\Sigma,t}$*.*

Proof of theorem 1. Immediate from Lemma 1 by expanding the definition of NI since e is a closed term. □

Lemma 1 (Typing soundness (generalized))**.** *If* $\overline{x : t} \vdash e : t$*,* $e[\overline{x} \mapsto \overline{v_1}] \longrightarrow^*_{\Omega_1} v_1'$*,* $e[\overline{x} \mapsto \overline{v_2}] \longrightarrow^*_{\Omega_2} v_2'$*,* $\Omega_1 \sim_{\Sigma} \Omega_2$ *and* $\overline{v_1} \sim \overline{v_2}$*, then* $v_1' \sim_t v_2'$*.*

Proof. Mutual induction over the typing derivation $\Gamma \vdash e : t$ and analogous statement for the quoted language. The full proof can be found in the full version of the paper. □

3. Implementation

Since F# contains an abundance of features not relevant to the current development we implement the language presented in Section 2, rather than attempting to enrich the F# implementation with security types. Our implementation compiles programs in this language to executable F# code. Given that the presented language is a subset of F#, the compilation consists mainly of removing level annotations in types in the program and establishing a connection to the database server.

This allows reusing the F# infrastructure for language-integrated query, as well as the improvements to this mechanism [16].

To simplify writing programs in the presented language, we implement a type inference algorithm supporting polymorphism for both levels and types. The basic approach that is used is based on constraint generation and unification [23]. For efficiency reasons the implementation is based on equality constraints, even though full inference would require inequality constraints. Interpreting inequality constraints as equality constraints introduces inaccuracies that prevents the types of some programs to be inferred properly.

However, since constraints are only generated in case they cannot be shown to be satisfied at the point of introduction it is always possible to resolve any such inaccuracies by providing type information in the form of type annotations. In practice, the type inference allows us to leave out many type annotations as witnessed by the examples in this paper.

The type-checker and compiler are implemented in Haskell, using the *BNFC* tool [3] for generating parsing and lexing code. The resulting binary takes a program in the language presented in Section 2.1 and produces F# code as output if the program is well-typed. If the program is not well-typed an error message detailing the reason for the type-checking failure is produced.

To illustrate the compilation, consider the output of the compiler for the example from Section 2.1 that queries the database for couples where the age difference between partners is greater than 10 years.

```
// import statements omitted

let ConnectionString_PeopleDB =
  "Data Source=.\MyInstance;Initial "\
  "Catalog=PeopleDB;Integrated Security=SSPI"
type dbSchema_PeopleDB =
  SqlDataConnection<ConnectionString_PeopleDB>
let db_PeopleDB = dbSchema_PeopleDB.GetDataContext()
let db = <@  db_PeopleDB  @>
type ResultType =  {name : string; diff : int; }
let differences : Expr<ResultType IQueryable> =
 <@ query { for c in (%db).Couples do
            for p1 in (%db).People do
            for p2 in (%db).People do
            if (c.Person1 = p1.Id) &&
               (c.Person2 = p2.Id) &&
               (abs (p1.Age - p2.Age) > 10) then
            yield { name = p1.Name
                  ; diff = p1.Age - p2.Age } }
 @>
let main = PLinq.Query.qquery
  { for x in (%differences) do yield x }
main
```

The above code example first imports all necessary libraries as well as the implementation of the supplementary concepts [16]. The subsequent part handles establishing a connection to the database server running on the same machine. The compiler generates a separate connection to the server for each database that is used by the program. Type synonyms and function definitions are compiled in a straight-forward way. The main difference is that all security levels have been removed from any types in the program.

For technical reasons, F# does not support query generation for quoted list expressions and therefore the compiler translates occurrences of the **list** type to **IQueryable** instead. Moreover, we translate expressions of the form **run** e into calls to a function `testPLinqQ` from the implementation accompanying [16]. This function takes a quoted expression, translates it into an SQL query, executes it and then returns the results.

Since our approach is purely static, and all security type information is erased during compilation, performance is unaffected, compared to ordinary F# code. Additionally, by reusing the results from Cheney et al. [16], we are able to benefit from the optimizations to F#'s LINQ mechanism presented there. Cheney et al. include a performance evaluation that is also valid for this implementation.

The code for the implementation is available online. The URL is given in Section 1.

4. Algebraic Data Types

We extend the language presented so far with algebraic data types and information-flow control for them.

This enriches the language with a way to express parametrized recursive data types that subsumes tuples and records. The addition is a proper extension to the language; neither tuples nor records can be recursive or parametrized in our language. We argue that introducing algebraic data types is a natural development due to their expressiveness and easy deconstruction via pattern matching. Encoding algebraic data types in an extended notion of records would both require extensions to the existing constructs that are similar to the extension needed to add algebraic data type and the result would be significantly less elegant.

Algebraic data types allow for the definition of new data types by composing existing data types. An algebraic data type consists of one or more constructors that can contain another type as their argument, including recursive occurrences of the defined data type. Pattern-matching is used deconstruct values in an algebraic data type by matching against the different constructors and parameters. The data contained in the parameters of a value in the data type can be extracted by giving a variable in the pattern.

Syntax Without loss of generality, consider an algebraic data type T with type argument α, which can be a product of several type variables. Constructors $C_1, \ldots, C_l$ have the form C_i of t_i where t_i is the argument of the constructor. Constructors with no arguments can be considered to take a value of unit type as an argument. For clarity, we only match on the outermost constructor of a single expression at a time. To track information flow, a security level annotation is then added to the type T. The expressions and values are extended as follows:

$$e ::= \ldots \mid C_i\ e \mid \textbf{match}\ e\ \textbf{with}\ C_1\ x_1 \to e_1\ ;\ \ldots\ ;\ C_k\ x_k \to e_k$$

$$\mathcal{E} ::= C_1\ \mathcal{E} \mid \ldots \mid C_k\ \mathcal{E} \mid \textbf{match}\ \mathcal{E}\ \textbf{with}\ C_1\ x \to e;\ \ldots\ ;\ C_k\ x \to e$$

$$\mathcal{Q} ::= C_1\ \mathcal{Q} \mid \ldots \mid C_l\ \mathcal{Q} \mid \textbf{match}\ \mathcal{Q}\ \textbf{with}\ C_1\ x \to e;\ \ldots\ ;\ C_k\ x \to e$$

$$V ::= \ldots \mid C_i\ V$$

Semantics The semantics is extended with the following rules for evaluation of constructors and pattern matching.

$$(\textbf{match}\ C_i\ v\ \textbf{with} \mid C_1\ x_1 \to e_1 \mid \ldots \mid C_k\ x_k \to e_k) \longrightarrow e_i[x_i \mapsto v]$$

These rules correspond to the usual semantics of algebraic data types in other functional languages. Constructors with values as arguments are themselves values and cannot be evaluated further. If a constructor argument is not a value, it is evaluated. **match** expressions evaluate the expression that is being matched on first, and then evaluate the appropriate branch while binding the argument to the constructor to a name.

Type system To support algebraic data types in the type system, we use two rule schemas which generate several typing rules for each algebraic data type in the program. For each algebraic data type with l constructors, one rule for **match** expressions is added and l typing rules for the constructors.

The rule schema for constructors takes into account that type arguments to constructors might contain the type that is being defined. In that case their level annotations need to be combined to keep the structure of the value confidential. $T^\ell \in t_i$ holds for all components of t_i of the for $\alpha\ T^\ell$.

In the case of **match** expressions, the structure of the algebraic data type is used to decide which branch to evaluate. To track this flow of information, the type of the branches needs to be upgraded to the level annotation of the algebraic data type. For this, we define an upgrade function $upg(t, \ell)$ which denotes upgrading the type t to have at least level ℓ in its outermost components.

Definition 4 (Upgrade function)**.** *$upg(t, \ell)$ is defined by recursion on the structure of t.*

$$
\begin{aligned}
upg(\textbf{\textit{int}}^{\ell}, \ell') &= \textbf{\textit{int}}^{\ell \sqcup \ell'} \\
upg(\textbf{\textit{bool}}^{\ell}, \ell') &= \textbf{\textit{bool}}^{\ell \sqcup \ell'} \\
upg(\textbf{\textit{string}}^{\ell}, \ell') &= \textbf{\textit{string}}^{\ell \sqcup \ell'} \\
upg(t \to t', \ell') &= t \to upg(t', \ell') \\
upg(t_1 * t_2, \ell') &= upg(t_1, \ell') * upg(t_2, \ell') \\
upg(\{\overline{f : t}\}, \ell') &= \{\overline{f : upg(t, \ell')}\} \\
upg((t\ \textbf{\textit{list}})^{\ell}, \ell') &= (t\ \textbf{\textit{list}})^{\ell \sqcup \ell'} \\
upg(\textbf{\textit{Expr}}\langle t \rangle, \ell') &= \textbf{\textit{Expr}}\langle upg(t, \ell') \rangle \\
upg((\alpha\ T)^{\ell}, \ell') &= (\alpha\ T)^{\ell \sqcup \ell'}
\end{aligned}
$$

$$
\textsc{Constr}\ \frac{e : t_i}{\Gamma \vdash C_i\ e : T^{\bigsqcup_{T^{\ell} \in t_i} \ell}}
$$

$$
\textsc{Match}\ \frac{\Gamma \vdash e : (\alpha\ T)^{\ell} \qquad \forall 1 \le i \le l. \Gamma, x_i : t_i \vdash e_i : t}{\Gamma \vdash (\textbf{match}\ e\ \textbf{with} \mid C_1\ x_1 \to e_1 \mid \ \ldots\ \mid C_k\ x_k \to e_k) : upg(t, \ell)}
$$

In the rule Constr, $\bigsqcup_{T^{\ell} \in t_i} \ell$ denotes the join of all levels on occurrences of T in the type t_i. This ensures that the level annotation on the resulting value is not lower than its components. For instance, the constructor rule for the node constructor of a binary tree type will require the structure of the constructed tree to be at least as confidential as the structure of the two sub-trees.

To be able to extend the soundness result for the type system to algebraic data types, the family of equivalence relations $\sim$ also needs to be extended for each algebraic data type. In doing so, we follow the intuition given for $\sim$ in the base language. The level annotation ℓ on $(\alpha\ T)^{\ell}$ corresponds to the confidentiality of the *structure* of the type, i.e. which constructor a value consists of. If ℓ is high, we consider the entire value, including components, to be confidential.

It should be pointed out that the rule schemas assume that the defined algebraic data types are well-formed, i.e.,

- recursive occurrences of the defined type must have the same type argument α, and
- the only type variables that can occur in arguments to constructors must be type variables in α.

Soundness The low-equivalence relation is extended to the values of algebraic data types. As for the built-in list data type, if $\ell = \mathtt{L}$, arguments to constructors may or may not be confidential, depending on their level annotations.

$$
\frac{\ell = \mathtt{L} \Rightarrow (i = j \wedge v_1 \sim_{t_i} v_2)}{C_i\ v_1 \sim_{\alpha\ T^{\ell}} C_j\ v_2}
$$

We prove the same soundness theorem as for the base language in this extended setting.

Theorem 2. *If $\vdash e : t$, $\Omega_1 \sim_{\Sigma} \Omega_2$, $e \longrightarrow^{*}_{\Omega_1} v_1$ and $e \longrightarrow^{*}_{\Omega_2} v_2$, then $v_1 \sim_t v_2$.*

Proof. Extension of proof for Lemma 1 for the new typing rules that are induced by algebraic data types. □

Note that while the theorem statement is the same, the set of types and expressions is now potentially larger, since it is extended in accordance with the algebraic data types defined in e.

Example: lists One common use for algebraic data types is to define recursive structures such as list. To demonstrate that our extension is capable of supporting such use cases, consider the following user-defined list data type:

```
type 'a MyList =
  | Nil
  | Cons of ('a, 'a MyList)
```

Instantiating the above rule schemas for the user-defined list type **MyList** yields the following three type rules; two for the constructors, and one for the matching.

$$
\frac{}{\Gamma \vdash \textbf{Nil} : {'a}\ \textbf{MyList}^{\ell}} \qquad \frac{\Gamma \vdash e_1 : {'a} \qquad \Gamma \vdash e_2 : {'a}\ \textbf{MyList}^{\ell}}{\Gamma \vdash \textbf{Cons}\ (e_1, e_2) : {'a}\ \textbf{MyList}^{\ell}}
$$

$$
\frac{\Gamma \vdash e : {'a}\ \textbf{MyList}^{\ell} \qquad \Gamma \vdash e_1 : t \qquad \Gamma, x : ('a, {'a}\ \textbf{MyList}^{\ell}) \vdash e_2 : t}{\Gamma \vdash \textbf{match}\ e\ \textbf{with} \mid \textbf{Nil} \to e_1 \mid \textbf{Cons}\ x \to e_2 : upg(t, \ell)}
$$

The generated rules match the intuitions given for the rest of the type system. Since **match** expressions information about the results of the branches (which have level ℓ') as well as the structure of the list (i.e. level ℓ) that the expression matches on, the level of the resulting list is $\ell \sqcup \ell'$. Moreover, the type system allows us to define corresponding functions for the **yield**, **exists**, @, and **for** constructs that are built into the language. The inferred type of each definition is given as a comment. Since the implementation sometimes generates extraneous type variables in inferred types that have no effect on generality, we give slightly simplified but equivalent types here.

```
// t → (t MyList)^ℓ
let yield' =
  fun x -> Cons (x, Nil)

// (t MyList)^ℓ → bool^ℓ
let exists' = fun xs -> match xs with
  | Nil -> False
  | Cons xs' -> True

// (t MyList)^ℓ → (t MyList)^ℓ → (t MyList)^ℓ
let rec union' = fun xs -> fun ys -> match xs with
  | Nil -> ys
  | Cons xs' -> Cons (fst xs', union' (snd xs') ys)

// (t1 MyList)^ℓ1 → (t1 → (t2 MyList)^(ℓ1⊔ℓ2))
// → (t2 MyList)^(ℓ1⊔ℓ2)
let rec for' =
  fun xs -> fun f -> match xs with
  | Nil -> Nil
  | Cons xs' -> union' (f (fst xs'))
                       (for' (snd xs') f)
```

Note that the types of these functions correspond roughly to the typing rules given for the built-in constructs. However, in the case of `union'` and `for'`, the type is slightly more restrictive than the typing rule, due to the way recursion is type-checked. However, these restrictions only affect the type of arguments and may only require lifting an argument expression to a higher security level.

Example: trees To further illustrate algebraic data types in the context of information flow, we discuss another common use, namely tree structures. We define an algebraic data type for binary trees:

```
type 'a BinTree =
  | Leaf
  | Node of (('a BinTree * 'a) * 'a BinTree)
```

In the same manner as for the user-defined list type, this will result in one rule for **match** expressions and two rules for the constructors:

$$\frac{}{\Gamma \vdash \textbf{Leaf} : (' a\ \textbf{BinTree})^{\ell}}$$

$$\frac{\Gamma \vdash e : ((('a\ \textbf{BinTree})^{\ell_1} * {}' a) * ('a\ \textbf{BinTree})^{\ell_2})}{\Gamma \vdash \textbf{Node}\ e : (' a\ \textbf{BinTree})^{\ell_1 \sqcup \ell_2}}$$

$$\frac{\Gamma \vdash e : (' a\ \textbf{BinTree})^{\ell_1 \sqcup \ell_2} \qquad \Gamma \vdash e_1 : t \qquad \Gamma, x : ((('a\ \textbf{BinTree})^{\ell_1} * {}' a) * ('a\ \textbf{BinTree})^{\ell_2}) \vdash e_2 : t}{\Gamma \vdash \textbf{match}\ e\ \textbf{with} \mid \textbf{Leaf} \rightarrow e_1 \mid \textbf{Node}\ x \rightarrow e_2 : upg(t, \ell_1 \sqcup \ell_2)}$$

The two typing rules for the constructors ensure that confidentiality of the tree structure is propagated correctly from the subtrees that are passed to the **Node** constructor. This construction is analogous to typing rules for lists in that the structure of the tree might be public while the tree elements might be confidential.

To illustrate the last point, consider a tree where the structure of the tree is not confidential while its elements are secrets:

```
let privTree : (int^H BinTree)^L =
  Node ((Leaf, (5 : int^H)),
        Node ((Leaf, (6 : int^H)), Leaf))
```

Since only the content at the leaves is considered private, counting the number of leaves of this tree can be typed with `L`:

```
let rec countLeaves =
  fun t -> match t with
    | Leaf -> 1
    | Node x -> countLeaves (fst (fst x)) +
                1 +
                countLeaves (snd x)

let result : int^L = countLeaves privTree
```

In contrast, trying to add all the integers in this tree and annotating the result with a low type will not type-check, since the computation involves more than merely the structure of the tree:

```
let rec sumElements =
  fun t -> match t with
    | Leaf -> 1
    | Node x -> sumElements (fst (fst x)) +
                snd (fst x) +
                sumElements (snd x)

// this is not well-typed:
let result' : int^L = sumElements privTree
```

5. Case Study: Movie Rental Database

In this section we exemplify the type system on a realistic example, a database to keep track of customer records by a movie rental chain, depicted in Figure 10. The example data and database schema [10] are courtesy of `postgresqltutorial.com` with permission to use their sample database in this work. The database contains information about approx. 16000 rentals, 600 customers, and 1000 movies. We use an existing sample database to demonstrate that our technique is applicable for database schemas that were not designed with information flow security in mind.

We first introduce a security policy for the database and consider various interesting queries that can be performed. Using the same setting, we illustrate the use of algebraic data types.

5.1 Basic Queries

The database keeps track of various information related to the movie rentals. Each rental is associated with a film, a customer, and a payment. The payments contain payment information and identifies the staff and the customer involved in the transaction. For both staff and customers address information is stored.

A reasonable security policy for such a database is to consider the names and exact addresses of customers and staff as confidential, while the rest of the data is considered public. In particular, the city of customers and the payment information are not considered confidential. The former is not a problem unless the city uniquely identifies a person and the latter does not contain any sensitive information. This security policy allows for querying the database for various interesting statistical information without disclosing confidential information about the customers.

Consider, for instance, the following example, which collects all rental ids for a given city.

```
let db = <@ database "Rentals" @>

let findCityId =
  <@ fun city -> for c in (%db).City do
                 if c.City1 = city then
                 yield c.City_id @>

let cityRentals : Expr<string^L -> int^L list^L> =
  <@ fun city -> for cid in (%findCityId) city do
                 for r in (%db).Rental do
                 for cu in (%db).Customer do
                 for a in (%db).Address do
                 if a.City_id = cid &&
                    cu.Address_id = a.Address_id &&
                    r.Customer_id = cu.Customer_id
                 then yield r.Rental_id @>
```

First in the example is the function `findCityId` that collects the city ids for a city of a given name. This function is used in `cityRentals` to look for rentals by customers living in that city. Note that while customer data is used, the type system ensures that only non-sensitive data affects the computation of the result. The rental ids can easily be used to produce interesting statistics about the relative popularity of films for different cities.

In contrast, trying to find all customers who rented a particular movie while forces the result to be secret, since the names of the customers are confidential. Thus, the following program is rejected by the type checker:

```
let rentalsForMovieTitle =
 <@ fun title -> for f in (%db).Film do
                 for r in (%db).Rental do
                 for i in (%db).Inventory do
                 if f.Title = title &&
                    r.Inventory_id = i.Inventory_id &&
                    i.Film_id = f.Film_id
                 then yield r @>

let customersWhoRented
      : Expr< string^L -> string^L list^L > =
 <@ fun title ->
      for r in (%rentalsForMovieTitle) title do
```

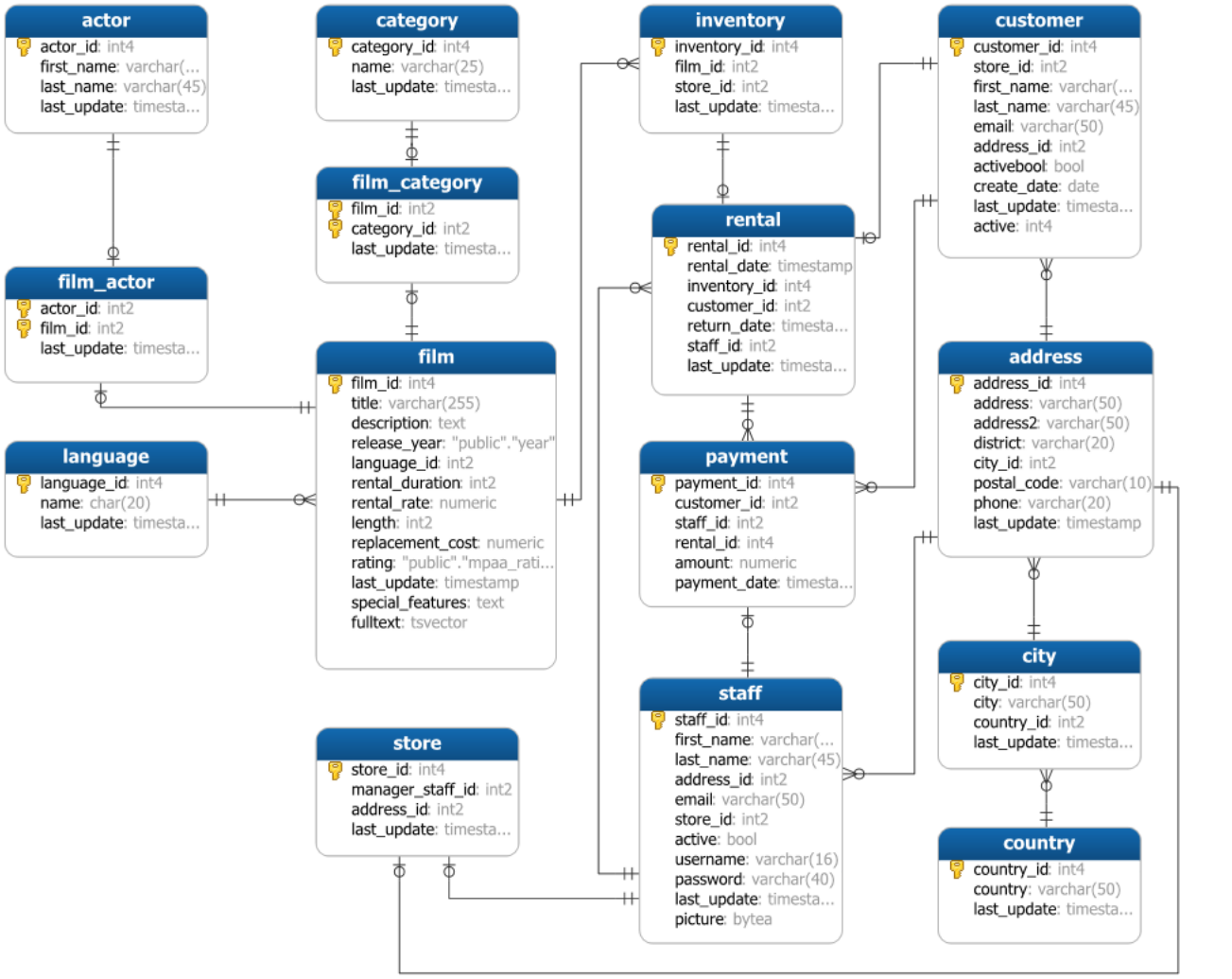

Figure 10. E-R diagram of movie rental database (image courtesy of `postgresqltutorial.com`)

```
    for c in (%db).Customer do
    if c.Customer_id = r.Customer_id
    then yield c.Last_name @>
```

The reason for the (correct) type error is that first and last names of customers are typed as **string**^{H} while the function `customersWhoRented` attempts to return a list containing elements of type **string**^{L}. Changing the security annotation to reflect this makes the type system accept the program.

More complicated queries can be handled with the same ease as the simpler above examples. Consider, for instance, the following query that finds all movies that were rented at least twice by the same customer:

```
let moviesRentedTwice : Expr< int^L list^L > =
 <@ for r1 in (%db).Rental do
   for r2 in (%db).Rental do
   for i in (%db).Inventory do
   for f in (%db).Film do
   for c in (%db).Customer do
   if not (r1.Rental_id = r2.Rental_id) &&
      r1.Inventory_id = i.Inventory_id &&
      r2.Inventory_id = i.Inventory_id &&
      i.Film_id = f.Film_id &&
      r1.Customer_id = c.Customer_id &&
      r2.Customer_id = c.Customer_id
   then yield f.Film_id @>
```

Thus, the above examples illustrate the power of the method clearly. By giving a security policy for the contents of the database we are able to track information flow in advanced queries in term of the information flow of the quoted language. Not only does this allow us to establish security information flow in programs that interact with databases, it does so in a way that is intuitively simple to understand; an additional benefit of expressing the database interaction in a homogeneous way is that it makes the information flow in the interaction more immediate.

5.2 Algebraic Data Types

To demonstrate the usefulness of information-flow tracking for algebraic data types discussed in Section 4 in a more practical setting, we will now consider an example demonstrating information-flow tracking from values stored in the database in conjunction with user-defined algebraic data types.

One plausible scenario for the use of such a database is to aggregate some information about the customer in order to make predictions about which movies he would be interested in. For

instance, one might want to determine a user's favorite category along with the movies he watched in that category. To that end, we can introduce the following algebraic data type that encodes a category along with a list of movie ids. (We only consider two categories for simplicity.):

```
type Category =
   | Action of (int^L list^L)
   | Scifi of (int^L list^L) ;;
```

Moreover, this information could be part of a larger record that stores information about a customer, where some information might be confidential and should not be used when the program output can be observed by the attacker. As an example, we consider a program that produces records of the following form:

```
type UserInfo =
  { uid : int^L
  ; firstName : string^H
  ; lastName : string^H
  ; favoriteCategory : Category^L
  } ;;
```

With this type, information about the favorite movie genre of a user can be used for prediction purposes, while the actual *name* of the customer cannot be retrieved without the resulting program being typed as H.

The following code produces a list of categories along with movie ids that a user, identified by their id, has rented:

```
let filmsByCustomer =
  <@ fun uid ->
     for r in (%db).Rental do
     for i in (%db).Inventory do
     for f in (%db).Film do
     if r.Customer_id = uid &&
        r.Inventory_id = i.Inventory_id &&
        i.Film_id = f.Film_id
     then yield f.Film_id @> ;;

let filmCategories =
  <@ fun fid ->
     for c in (%db).Category do
     for cf in (%db).Film_category do
     if cf.Film_id = fid &&
        cf.Category_id = c.Category_id
     then yield c.Name @> ;;

let userMovieInfo =
  <@ fun uid ->
     for fid in (%filmsByCustomer) uid do
     for cname in (%filmCategories) fid do
     yield { catname = cname ; fid = fid } @> ;;
```

Since the LINQ framework in F# does not allow producing values of user-defined algebraic data types from within a query, we first need to produce a record that contains the list of categories and movie ids returned by `userMovieInfo`.

```
let compileStats : Expr< UserInfo list^L > =
  <@ for cust in (%db).Customer do
     yield { uid = cust.Customer_id
           ; firstName = cust.First_name
           ; lastName = cust.Last_name
           ; movieCategories =
             (%userMovieInfo) cust.Customer_id } @>
```

To turn the information in the `movieCategories` field into an element of the defined algebraic data type, the following code counts the given list of movie data and then produces a value of type **Category** depending on which category occurs more often. (This is intentionally not written in a functional style to avoid having to introduce many additional auxiliary functions commonly found in functional languages.) The code then constructs a new record with the `movieCategories` replaced by the users favorite category.

Note that this computation now takes place in the host language, and security levels from the query result are propagated to these functions.

```
let updateCount =
  fun minfo -> fun statsrec ->
    { actionMovies =
      if' (minfo.catname = "action")
          (yield minfo.fid) [] @
      statsrec.actionMovies
    ; scifiMovies =
      if' (minfo.catname = "scifi")
          (yield minfo.fid) [] @
      statsrec.scifiMovies }

let emptyCounts = { actionMovies = [] ; scifiMovies = [] }

let countCategories =
  fun catList -> fold catList updateCount emptyCounts

let favoriteUserCategory =
  fun minfos ->
  let statsrec = countCategories minfos
  in (if' (length statsrec.actionMovies >
           length statsrec.scifiMovies)
          (Action (statsrec.actionMovies))
          (Scifi (statsrec.scifiMovies)))

let stats = map (fun x ->
  { uid = x.uid
  ; firstName = x.firstName
  ; lastName = x.lastName
  ; favoriteCategory =
    favoriteUserCategory (x.movieCategories) })
  (run compileStats)

let getCategories : Category^L list^L =
  map (fun x -> favoriteUserCategory
                  (x.movieCategories))
      (run compileStats)
```

The type system then correctly infers that the computation of the category does in fact not depend on confidential information about the user, while the name and email fields of the resulting records do. `if'` works like the built-in **if** construct except that it can produce values that are not lists and also requires an expression for the else case.

Moreover, attempting to find the favorite category of one particular user, identified by name, and typing the result with L will be prevented by the type checker. Concretely, an example such as the following, will be rejected by the type checker:

```
let attack : Category^L list^L =
  for x in getCategories do
  if x.firstName = "John" && x.lastName = "Doe"
  then yield x.favoriteCategory
```

6. Related Work

Until recently, little work has been done on bridging information-flow controls for applications [13, 29, 30, 41] and databases they manipulate. While mainstream database management systems such as PostgreSQL [7], SQLSever [8], and MySQL [6] include pro-

tection mechanisms at the level of table and columns, as is, these mechanisms are decoupled from applications.

Below, we focus on the work that shares our motivation of integrating the security mechanisms of the application and database, with the goal of tracking information flow.

WebSSARI by Huang et al. [32] is a tool that combines static analysis with instrumented runtime checks. The focus is on PHP applications that interact with an SQL database. The system succeeds at discovering a number vulnerabilities in PHP applications. Given its complexity, its soundness is only considered informally.

Li and Zdancewic [35] present an imperative security-typed language suitable for web scripting and a general architecture that includes a data storage, access control, and presentation layers. The focus is on suitable labels for confidentiality and integrity policies as well as the possibilities of safe label downgrading [44]. No soundness results for the type system are reported.

A line of work has originated from, or influenced by, from Links by Cooper et al. [21], a strongly-typed multi-tier functional language for the web. Links supports higher-order queries. On the other hand, Links comes with a non-standard database backend, making its interoperability non-trivial.

DIFCA-J by Yoshihama et al. [53] is an architecture for dynamic information-flow tracking in Java. The architecture covers database queries as performed by Java programs via Java DataBase Connectivity (JDBC) APIs.

Baltopoulos and Gordon [12] study secure compilation by augmenting the Links compiler with encryption and authentication of data stored on the client. Source-level reasoning is formalized by a type-and-effect system for a concurrent λ-calculus. Refinement types are used to guarantee that integrity properties of source code are preserved by compilation.

SELinks by Corcoran et al. [22] also builds on Links. With the Fable type system by Swamy et al. [49] at the core, the authors study the propagation of labels, as described by user-defined functions, through database queries. Fable's flexibility accommodates a variety of policies, including dynamic information-flow control, provenance, and general safety policies based on security automata.

DBTaint by Davis and Chen [24] shows how to enhance database data types with one-bit taint information and instantiate with two example languages in the web context: Perl and Java.

Chlipala's UrFlow [18] offers a static information-flow analysis as part of the Ur/Web domain-specific language for the development of web applications. Policies can be defined in terms of SQL queries. User-dependent policies are expressed in terms of the users' runtime knowledge.

Caires et al. [15] are interested in type-based access control in data-centric systems. They apply refinement types to express permission-based security, including cases when policies dynamically depend on the state of the database. This line of work leads to information-flow analysis by Lourenço and Caires [37]. This analysis is presented as a type system with value-indexed security labels for λ-calculus with data manipulation primitives. The type system is shown to enforce noninterference.

Hails by Giffin et al. [27] is a web framework for building web applications with mandatory access control. Hails supports a number of independently such useful design pattern as privilege separation, trustworthy user input, partial Lourenço and Caires [37] update, delete, and privilege delegation.

IFDB by Schultz and Liskov [46] proposes a database management system with decentralized information-flow control. IFDB is implemented by modifying PostgreSQL as well as modifying application environments in PHP and Python. The underlying model is the Query by Label model that provides abstractions for managing information flows in a relational database. This powerful model includes confidentiality and integrity labels, and models decentralization and declassification.

LabelFlow by Chinis et al. [17] dynamically tracks information flow in PHP. It is designed to deal with legacy applications, and so it transparently extends the underlying database schema to associate information-flow labels with every row.

The SLam calculus by Heintze and Riecke [31] pioneers information-flow control in a functional setting. The security type system treats a simple language with first-class functions, based on the λ-calculus. This is the first illustration of how noninterference can be enforced in the functional setting. Our security type system adopts as the starting point the security type system by Pottier and Simonet [40], which they have developed for a core of ML, and which serves as the base for the Flow Caml tool [48]. Compared to that work, our system includes the formalization and implementation of algebraic data types and pattern matching. Experiments with Flow Caml indicate support for algebraic data types but without evidence of soundness [40].

The tools like SIF [19], SWIFT [20], and Fabric [36] allow the programmer to enforce powerful policies for confidentiality and integrity in web applications. The programmer labels data resources in the source program with fine-grained policies using Jif [38], an extension of Java with security types. The source program is compiled against these policies into a web application where the policies are tracked by a combination of compile-time and run-time enforcement. The ability to enforce fine-grained policies is an attractive feature. At the same time, SIF and SWIFT do not provide database support. Fabric supports persistent storage while leaving interoperability with databases for future work.

A final note on related work is that care has to be taken when setting security policies for sensitive databases. Narayanan and Shmatikov's widely publicized work [39] demonstrates how to de-anonymize data from Netflix' database (where names were "anonymized" by replacing them with random numbers) using publicly available external information from sources as the Internet Movie Database [9].

7. Conclusion

We have presented a uniform security framework for information-flow control in a functional language with language-integrated queries (with Microsoft's LINQ on the backend). Because both the host language and the embedded query languages are both functional F#-like languages, we are able leverage information-flow enforcement for functional languages to obtain information-flow control for databases "for free", synergize it with information-flow control for applications, and thus guarantee security across application-database boundaries . We have developed a security type system with a novel treatment of algebraic data types and pattern matching, and established its soundness. We have implemented the framework and demonstrated its usefulness in a case study with a realistic movie rental database.

A natural direction for future work includes support of declassification [44] policies. This will enable more fine-grained labels and richer scenarios with intended information release. The functional setting allows for particularly smooth integration of policies of *what* [34, 35, 43] is released, where we can express aggregates through *escape hatches* [42], as represented by functions with no side effects. We believe that enriching the model with these policies will also open up for direct connections to the *database inference* [26] problem, much studied in the area of databases.

Acknowledgments

Thanks are due to Phil Wadler whose talk on language-integrated queries at Chalmers was an excellent inspiration for this work. This

work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

References

[1] SPARKAda Examinar. Software release. http://www.praxis-his.com/sparkada/.

[2] OWASP Top 10: Ten Most Critical Web Application Security Risks. https://www.owasp.org/index.php/Top_10_2013-Top_10/, 2013. Accessed: 2014-02-20.

[3] BNF Converter. http://bnfc.digitalgrammars.com/, 2014. Accessed: 2014-02-20.

[4] Google Web Toolkit. http://www.gwtproject.org/, 2014. Accessed: 2014-02-20.

[5] LINQ (Language-Integrated Query). http://msdn.microsoft.com/en-us/library/bb397926.aspx, 2014. Accessed: 2014-02-20.

[6] Privileges Provided by MySQL. https://dev.mysql.com/doc/refman/5.1/en/privileges-provided.html, 2014. Accessed: 2014-02-20.

[7] Database Roles and Privileges. http://www.postgresql.org/docs/9.0/static/user-manag.html, 2014. Accessed: 2014-02-20.

[8] Authorization and Permissions in SQL Server. http://msdn.microsoft.com/en-us/library/bb669084(v=vs.110).aspx, 2014. Accessed: 2014-02-20.

[9] Internet Movie Database. http://www.imdb.com/, 2014. Accessed: 2014-02-20.

[10] PostgreSQL sample database. http://www.postgresqltutorial.com/postgresql-sample-database/, 2014. Accessed: 2014-02-20.

[11] Ruby on Rails. http://rubyonrails.org/, 2014. Accessed: 2014-02-20.

[12] I. G. Baltopoulos and A. D. Gordon. Secure compilation of a multi-tier web language. In *TLDI*, pages 27–38, 2009.

[13] N. Bielova. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *J. Log. Algebr. Program.*, pages 243–262, 2013.

[14] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying Facets of Information Integrity. In *ICISS*, pages 48–65, 2010.

[15] L. Caires, J. A. Pérez, J. a. C. Seco, H. T. Vieira, and L. Ferrão. Type-Based Access Control in Data-Centric Systems. In *ESOP*, pages 136–155, 2011.

[16] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, pages 403–416. ACM, 2013.

[17] G. Chinis, P. Pratikakis, S. Ioannidis, and E. Athanasopoulos. Practical information flow for legacy web applications. In *ICOOOLPS*, pages 17–28, 2013.

[18] A. Chlipala. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *OSDI*, pages 105–118, 2010.

[19] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proc. USENIX Security Symposium*, pages 1–16, Aug. 2007.

[20] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Building secure web applications with automatic partitioning. *Commun. ACM*, 52(2):79–87, 2009. .

[21] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *FMCO*, pages 266–296, 2006.

[22] B. J. Corcoran, N. Swamy, and M. W. Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD Conference*, pages 269–282, 2009.

[23] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212. ACM, 1982.

[24] B. Davis and H. Chen. DBTaint: Cross-application Information Flow Tracking via Databases. In *WebApps*, pages 12–12. USENIX Association, 2010.

[25] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[26] J. Domingo-Ferrer, editor. *Inference Control in Statistical Databases, From Theory to Practice*, volume 2316 of *LNCS*, 2002. Springer.

[27] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *OSDI*, pages 47–60, 2012.

[28] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. IEEE SP*, pages 11–20, Apr. 1982.

[29] G. L. Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[30] D. Hedin and A. Sabelfeld. A perspective on information-flow control. *Proc. of the 2011 Marktoberdorf Summer School. IOS Press*, 2011.

[31] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *POPL*, pages 365–377, 1998.

[32] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, pages 40–52, 2004.

[33] A. Kennedy. Types for Units-of-Measure: Theory and Practice. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *CEFP*, volume 6299 of *Lecture Notes in Computer Science*, pages 268–305. Springer, 2009. ISBN 978-3-642-17684-5. URL http://dblp.uni-trier.de/db/conf/cefp/cefp2009.html#Kennedy09.

[34] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, pages 158–170, 2005.

[35] P. Li and S. Zdancewic. Practical Information-flow Control in Web-Based Information Systems. In *CSFW*, 2005. .

[36] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: a platform for secure distributed computation and storage. In *SOSP*, pages 321–334, 2009.

[37] L. Lourenço and L. Caires. Information Flow Analysis for Valued-Indexed Data Security Compartments. In *TGC*, 2013.

[38] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[39] A. Narayanan and V. Shmatikov. Robust De-anonymization of Large Sparse Datasets. In *IEEE Symp. on Security and Privacy*, 2008.

[40] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL*, pages 319–330. ACM, 2002.

[41] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, pages 5–19, 2003.

[42] A. Sabelfeld and A. C. Myers. A Model for Delimited Information Release. In *ISSS*, volume 3233 of *LNCS*, pages 174–191, 2003.

[43] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *Higher Order and Symbolic Computation*, 14 (1):59–91, Mar. 2001.

[44] A. Sabelfeld and D. Sands. Declassification: Dimensions and Principles. *J. Computer Security*, 17(5):517–548, Jan. 2009.

[45] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Trans. Comput. Syst.*, pages 277–288, 1984.

[46] D. A. Schultz and B. Liskov. IFDB: decentralized information flow control for databases. In *EuroSys*, pages 43–56, 2013.

[47] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1. . URL http://dx.doi.org/10.1109/SP.2010.26.

[48] V. Simonet. The Flow Caml system. Software release. Located at http://cristal.inria.fr/~simonet/soft/flowcaml, 2003.

[49] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A Language for Enforcing User-defined Security Policies. In *IEEE Symp. on Security and Privacy*, 2008.

[50] D. Syme. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution. In *Workshop on ML*, pages 43–54. ACM, 2006. .

[51] D. Volpano. Safety versus Secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, Sept. 1999.

[52] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *J. Computer Security*, 4(3):167–187, 1996.

[53] S. Yoshihama, T. Yoshizawa, Y. Watanabe, M. Kudo, and K. Oyanagi. Dynamic Information Flow Control Architecture for Web Applications. In *ESORICS*, pages 267–282, 2007.