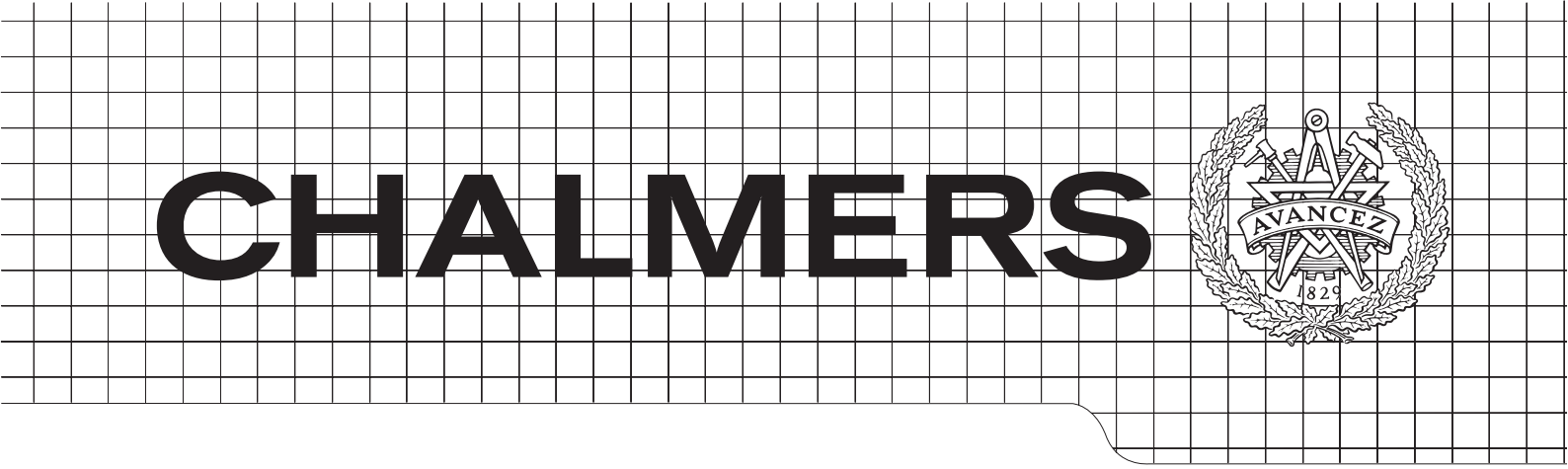

Information Flow in Databases for Free

Master's Thesis in Computer Science & Engineering

DANIEL SCHOEPE

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2014

Daniel Schoepe

Examiner: Andrei Sabelfeld
Supervisor: Andrei Sabelfeld

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden April 2014

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 work proposes a framework for uniform tracking of information flow through both the application and the underlying database. Key enabler of the uniform treatment is work by Cheney et al., presented at last year's ICFP, 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 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 end-to-end security. We develop the formal results in the form of a security type system that includes a novel 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.

Contents

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.

1.1 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.

It is not surprising that systems often break at component boundaries. The OWASP Top 10 project identifies the ten most critical web application security risks [OWA13]. The top of the list is dominated by attacks across component boundaries: injection attacks (with SQL injection as the 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* [SRC84], overarching component boundaries.

1.2 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.

Since applications often legitimately require access to both confidential information and the internet in order to perform their function, traditional access control mechanism cannot provide the necessary granularity for ensuring that privacy expectations of users are respected. As an example, consider a spreadsheet application that a user uses to manage their finances. Such an application requires access to confidential data belonging to the user, namely financial records, in order to perform its function. Moreover, the spreadsheet application requires access to the internet to look up information such as stock prices.

Another example of this is web applications; web applications are frequently built using code coming from different sources in addition to ads with executable content. While the application itself may come from a trusted source, included code in general and ads in particular cannot necessarily be trusted. For smart phones and tablets there are similar problems. In addition, in this setting the apps available from one of the many app stores are published with little or no verification. This opens up an easy way to inject malicious code onto such devices.

Nevertheless, many seem to be inclined to trust such programs with access to our sensitive information either directly or indirectly by allowing them to run on out devices. The results are similar across all platforms: successful targeted attacks with malicious code compromise sensitive information with direct negative effects on the users.

In this scenario traditional access control is of no use. The software must be granted certain access and execution rights for proper functionality. Of equal importance to access control is what the software does with its rights after permission has been granted. To provide *end-to-end* security information flow security tracks any sensitive information a program is given access to as it flows through the program during execution in order to prevent the information from being divulged to unauthorized parties.

The relevant question is therefore not whether an application should be granted access to data, but rather whether the *information flow* within the application conforms to the user's expectations. In the example of a spreadsheet application, the application should not disclose financial information about the user to any server on the internet.

Noninterference One way to make this notion more precise is to require that *private inputs* do not affect *public outputs*, i.e. confidential Information provided by a user should not influence the part of the behavior of a program that is observable by an attacker. Concretely, this could be the content of local files on a user's computer being transmitted over the internet to an untrusted server. If a user is unaware of this and does not want this data to be transmitted, it presents a breach of the user's privacy expectations.

This notion was first formalized by Goguen and Meseguer [GM82b] under the name *non-inteference*. This formalizes the aforementioned intuition in a language-agnostic way. When considering a concrete language, we need to state this property in terms of the semantics of the language under considerations. As mentioned, we follow standard approaches in the case of functional languages, such as the one by Pottier et. al. [PS02].

Information-flow control mechanisms need to take into account both *explicit* and *implicit* flows. The following informal examples will give an intuitive explanation of this distinction. For simplicity we avoid giving formal meaning to the used pseudo-code and noninterference notion. A precise definition for the developments in this work can be found in Section 2.

An *explicit* flow leaks private information directly over a public channel. As an example for an explicit flow, consider the following piece of pseudo-code:

```
secret = read_input(secret_channel)
send(public_channel, secret)
```

In this example we assume that `read_input` reads an input value from a channel given as an argument and `send` sends out information over a channel given as the first argument. In this case, the outputs on `public_channel` (e.g. a connection to an untrusted web server) directly depends on inputs on `secret_channel` (e.g. a private file on the hard disk). Hence, noninterference is violated.

An *implicit* flow leaks information through the control flow of a program, even when the secret itself is not sent over a public channel. Consider the following example where `secret` is assumed to be a confidential boolean value:

```
secret = read_input(secret_channel)
if secret
   then send(public_channel, 1)
   else send(public_channel, 0)
```

In this case, only constants are sent over a public channel, but the output on `public_channel` still depends on secret inputs. Hence, this program also violates noninterference.

In addition, another side of the increased use of personal electronics is data aggregation. Our use of devices for, for instance, communication, documentation, recreation and purchases leaves a digital trail with a potential monetary value. There is an ongoing debate on the dangers of this data aggregation. Regardless, the functionality of many services require some data to be retained, typically in some form of database, in order to be accessed at some later point.

In order to successfully guarantee end-to-end security in this setting it is paramount that the information-flow tracking crosses the boundary between the program itself and external services, like databases. If this is not done, there is a risk that information can be laundered. With this in mind and the ubiquity of aggregation of sensitive data in online databases it is somewhat surprising that relatively little work has been done on information flow security and databases.

Since the introduction of noninterference, a large, extensively surveyed [SM03b, Gue07, HS11, Bie13], body of work has studied information-flow control. However, with a few recent exceptions (discussed in Section 7), 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 work focuses on end-to-end information-flow control for systems with database components.

1.3 Programming Language Semantics

The work presented here falls within the category of *language-based* security, meaning that we try to enforce security properties by designing languages such that all programs written in it are secure *by construction*. For such results to be reliable, it is paramount that such security claims are proven to be true.

To achieve guarantees of this sort, it is necessary to specify precisely, using mathematical concepts, how a program is evaluated. This makes the language amenable to mathematical reasoning and hence allows to prove general statements about all programs that can be written in the language.

There are three major approaches to specifying programming language semantics: *Operational semantics* specify the meaning of programs by describing how a program is evaluated to value. *Denotational semantics* directly assign a mathematical object to

each program in a language. *Axiomatic semantics* specifies the laws of a programming language in terms of what can be proven about programs. For more background on programming language semantics and different styles of reasoning about them, the reader is referred to [Pie02].

In this work we use *operational semantics* to reason about the language introduced in Section 2. In particular, our semantics will be given as *small-step semantics*. This means that the meaning of programs is described by how bigger terms are changed into smaller terms in a step-by-step fashion. For example, in a language featuring addition and multiplication, the evaluation of an expression like $add(12, mult(5, add(7, 3)))$ can be divided into the following steps:

$$add(12, mult(5, add(7, 3))) \rightsquigarrow add(12, mult(5, 10)) \rightsquigarrow add(12, 50) \rightsquigarrow 62$$

Where $\rightsquigarrow$ is the relation that describes when one term evaluates to another. In this example, the relation can be specified as an *inductively defined set* by the following rules:

$$\frac{value(a) \qquad value(b)}{add(a,b) \rightsquigarrow a + b} \qquad \frac{value(a) \qquad b \rightsquigarrow b'}{add(a, b) \rightsquigarrow a + b'} \qquad \frac{a \rightsquigarrow a'}{add(a', b)}$$

$$\frac{value(a) \qquad value(b)}{mult(a,b) \rightsquigarrow a \times b} \qquad \frac{value(a) \qquad b \rightsquigarrow b'}{mult(a, b) \rightsquigarrow a + b'} \qquad \frac{a \rightsquigarrow a'}{mult(a', b)}$$

$value(a)$ denotes that a has been fully evaluated to an integer and is not a term such as $add(3, 5)$. Also note that $add(a, b)$ (resp. $mult(a,b)$) is a syntactic term, while $a + b$ (resp. $a \times b$) denotes ordinary mathematical addition (resp. multiplication). Moreover, the above rules enforce that the first argument of an operation is evaluated before the second. This is desirable in languages such as C, where expressions can have side effects, making the order of evaluation relevant.

Evaluation contexts One common concept to simplify the construction of these relations is called *evaluation contexts*. Intuitively, evaluation contexts can be thought of as "expressions with a hole in them", which will then be substituted for another expression.

As an example, note that the previous example contains some amount of duplication in the rules, since for both addition and multiplication, rules are need to allow evaluation "inside" expressions of the form $add(a, b)$ or $mult(a, b)$. This quickly becomes cumbersome as the number of features in a language increases.

To alleviate this problem in this example, we define evaluation contexts as terms of the following grammar:

$$\mathcal{E} ::= \bullet \mid add(\mathcal{E}, a) \mid add(v, \mathcal{E}) \mid mult(\mathcal{E}, a) \mid mult(v, \mathcal{E})$$

$\bullet$ represents the "hole" in the expression and v must be a fully evaluated value whereas a can be any expression. For an evaluation context $\mathcal{E}$, we denote replacing $\bullet$ by some term a with $\mathcal{E}[a]$.

Using this construction we can then express the same semantics more concisely:

$$\textsc{Add}\ \frac{value(a) \qquad value(b)}{add(a,b) \rightsquigarrow a + b} \qquad \textsc{Mult}\ \frac{value(a) \qquad value(b)}{mult(a,b) \rightsquigarrow a \times b} \qquad \textsc{Cxt}\ \frac{a \rightsquigarrow a'}{\mathcal{E}[a] \rightsquigarrow \mathcal{E}[a']}$$

Using these rules, we can evaluate components of more complex expressions using the third rule with a suitable evaluation context $\mathcal{E}$.

One step in the evaluation of the example $add(12, mult(5, add(7, 3)))$ can then be performed by setting $\mathcal{E} = add(12, mult(5, \bullet))$ and then applying the rules as in the following derivation:

$$\textsc{Cxt}\ \frac{\textsc{Add}\ \dfrac{value(7) \qquad value(3)}{add(7,3) \rightsquigarrow 10}}{(add(12, mult(5, \bullet)))[add(7,3)] \rightsquigarrow (add(12, mult(5, \bullet)))[10]}$$

Note that $(add(12, mult(5, \bullet)))[10]$ is by definition equal to $add(12, mult(5, 10))$ and hence this step exactly matches the evaluation without evaluation contexts. Moreover, due to the construction of evaluation contexts, this enforces the same evaluation order.

Following [CLW13], evaluation contexts are used in Section 2.1 to present the language semantics in a more compact form.

1.4 Type Systems

Type systems present an established way to statically ensure the absence of some classes of errors in programs. Simple examples include adding strings and integers or passing more parameters to a function than the function expects. More concisely, Pierce [Pie02] phrases the purpose of type systems as follows:

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.
>
> Benjamin Pierce, "Types for Programs and Proofs", [Pie02]

Static type checking has been employed in practice in a variety of programming languages such as C, Java, or Haskell. In sufficiently expressive type systems, types can even express arbitrary mathematical statements and type checking amounts to verifying a proof of these statements, as shown by Curry and Howard [Cur34, How].

Type systems are usually specified as *inductively defined relations* between a *typing context*, a *term* and a *type*. This is often written as $\Gamma \vdash e : t$, denoting that e has type t in context Γ. A *typing context* usually assigns types to free variables that may occur

in the expression e. Such variables are typically introduced by programs that define functions which take arguments that then occur in the body of the program.

To illustrate this, the type system for the simple arithmetic example from the previous section could include rules such as the following:

$$\frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}} \qquad \frac{\Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 \times e_2 : \textbf{int}}$$

These two rules state that if both operands of an addition (or multiplication) are integers, then the resulting addition (or multiplication) will result in an integer as well.

A type-checker will then check that a program can be assigned a type using such typing rules. If this is not possible (for example if the programmer tries to multiply a string and a number), the program is rejected. Again we refer the reader to [Pie02] for a more thorough introduction to type systems.

Security type systems While type systems were first thought of to provide an efficient way of ensuring certain *safety* properties of programs, such as avoiding that the program enters an undefined state during execution, they can also be used to ensure *security* properties, such as noninterference.

Such approaches have been used in the context of functional languages, for example by Pottier et. al. [PS02], but also for imperative settings (e.g. [MSS11, SM03b]).

One main advantage of using type systems to ensure that security properties hold, is that they can often be implemented using traditional type checking techniques, increasing confidence in the faithfulness of the implementation.

Our work follows Pottier et. al. [PS02] and similar approaches in the sense that the type system ensures both, safety properties, such as only applying a function to arguments of the correct type, as well as enforcing information-flow policies.

1.5 Database Integration

Programs commonly access databases via libraries that connect and interact with a 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();
```

While this is a powerful approach, it comes with inherent shortcomings. First, it offers no support in the construction of queries. There is no guarantee that the string is a valid SQL query or that the string has not been a result of an SQL injection. Second, 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 mechanisms into the language as facilitated, e.g, by Google's Web Toolkit [GWT14], Ruby on Rails [rub14], and Microsoft's LINQ [LIN14].

In a 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. [CLW13], presented at last year's ICFP. 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 $\mathbf{Expr}\langle t\rangle$. Antiquotes `(% )` provide a way to splice in other typed quoted values into other quoted expression. This approach capitalizes on the flexible meta-programming capabilities of F# [Sym06]. 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
```

The quoted expression is parsed by the F# runtime and the typed result is passed to *run* for normalization and evaluation, which 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.

1.6 Algebraic Data Types

A feature that is often provided by functional programming languages such as ML or Haskell, is the ability to define new data types in programs, to allow the user to write more high-level programs.

One particularly flexible method of supporting user-defined data types is known as *algebraic data types*. An algebraic data type consists of one or more constructors, each of which may take other values as arguments. Moreover, an algebraic data type can depend on *type variables* that can occur in the type of arguments to a constructor. The concept can be illustrated more clearly with a few simple examples.

For instance, algebraic data types allow the programmer to define his own boolean data type consisting of the values $True$ and $False$:

```
type Bool = True | False
```

A value of the newly defined type $Bool$ can be either the constructor $True$ or the constructor $False$. However, algebraic data types are more general than simple enumerations. For example, a user could define his own list data type in the following fashion:

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

In this example, the type $MyList$, taking a type variable $'a$ as an argument, represents lists with elements of type $'a$. For instance, **int** $MyList$ would represent a list of integers.

Nil represents the empty list. $Cons\ (x, xs)$ represents a list the first element of which is x and the rest of the list consists of the list xs.

While algebraic data types are a well-known concept in functional programming, they are rarely considered in the context of information flow. Section 4 presents a treatment of algebraic data types in the context of information-flow control.

More detailed introductions to algebraic data types can be found in language documentation for languages that support them, for instance in an introduction to Haskell by Hudak et. al. [HF92].

1.7 Contributions

The work by Cheney et al. is a key enabler for enforcing end-to-end security in a uniform functional setting. 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 end-to-end security.

In a nutshell, the thesis contains the following contributions:

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

(ii) We develop the formal results in the form of a security type system and show that it enforces the security condition of *noninterference* [GM82a] (Section 2).

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

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

(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.

[1] `http://schoepe.org/~daniel/selinq/`

2.1 Language

The language is based on the one used by Cheney et al. [CLW13] 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.

$\ell ::=$ '`L`' | '`H`'

$b ::=$ '`int`'$^\ell$ | '`string`'$^\ell$ | '`bool`'$^\ell$

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

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

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

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

Figure 1: Syntax of language and types

We now give some intuition for the language constructs:

We assume that there is a predefined set of constants and operators that is built into the language. These can include constants such as integers and booleans, and operators such as addition and multiplication of integers, comparison operations, or string concatenation. Expressions of the form $op(\overline{e})$ denote applying such a built-in operator op to a sequence of arguments $\overline{e}$.

Expressions of the form $\mathbf{fun}(x) \to e$ allow constructing a function taking an argument named x with function body e, where x may occur free in e. Functions with multiple arguments are constructed by using **fun** repeatedly. For example, a function taking arguments x and y and returning their sum, can be expressed by $\mathbf{fun}(x) \to \mathbf{fun}(y) \to x + y$. $\mathbf{rec}\ f(x) \to e$ defines a recursive function f, taking an argument x. In contrast to **fun** expressions, the name of function (i.e. f) that is being constructed can occur in the body e.

(e_1, e_2) denotes a tuple consisting of expressions e_1 and e_2, while $\mathbf{fst}\ e$ and $\mathbf{snd}\ e$ allow deconstructing tuples to retrieve their components.

Records contain named fields $f_1, \ldots, f_n$ and are constructed by assigning expressions $e_1, \ldots, e_n$ to the fields, i.e. by writing $\{f_1 = e_1, \ldots, f_n = e_n\}$. Fields can be accessed by their name f, using expressions of the form $e.f$.

Lists are constructed using [], **yield**, and **union**, where [] denotes the empty list,

yield e constructs a singleton list containing only the expression e and e_1 @ e_2 concatenates lists e_1 and e_2. **exists** e evaluates to **true** if and only if the expression e does not evaluate to the empty list and 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 **true** and to [] otherwise. **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 are then concatenated.

run e denotes running a quoted expression e. This usually 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. `<@` 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 an antiquotation of the expression e, and allows splicing of quoted expressions into quoted expressions in a type-safe way. E.g. the following two programs yield the same result:

```
let x = <@ 5 @>
let y = <@ fun z -> 7 + z * (%x) @>
// yields the same result:
let y = <@ fun z -> 7 + z * 5 @>
```

Security type language The security type language is defined by annotating a standard type language for a functional fragment with 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 [BRS10]. 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 in this setting, we consider a database to be a collection of tables. Each table consists of at least one named column. Novel to our setting are security annotations on columns: each column has a fixed type which is also annotated with a security level. 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 the example by Cheney et al. [CLW13] 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 : int^L; Name : string^H; Age : int^L } list^L
  ; Couples : { Him : int^L ; Her : int^L } list^L }
```

Now consider the situation where we want to query the database for couples where the woman is older than her 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 woman is older than the man. If that is the case, the name of the woman 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 : string^H ; diff : int^L}

let differences : Expr < ResultType list ^ L > =
  <@ for c in (% db).Couples do
     for w in (% db).People do
     for m in (% db).People do
     if (c.Her = w.Id) && (c.Him = m.Id) &&
        (w.Age > m.Age) then
     yield ({ name = w.Name; diff = w.Age - m.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 as 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 follows results in a type error.

```
// No longer well-typed:
type ResultType = {name : string^L ; diff : int^L}
```

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, for example mapping an addition operator in the language to ordinary mathematical addition.

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

Figure 2: Values and evaluation contexts

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

The evaluation rules in Figures 2, 3, 4, and 5 follow [CLW13]. 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. We define $eval_\Omega(e) = v$ if $e \longrightarrow_\Omega^* v$, where v is a value; similarly we define $norm(e) = v$ if $e \rightsquigarrow^* e'$ and $e' \hookrightarrow^* v$.

Details and properties of this evaluation can be found in [CLW13].

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 $e[x \mapsto e']$.

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

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

Figure 3: Evaluation rules for host language

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

Figure 4: Symbolic reduction phase

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

Figure 5: Ad-hoc reduction phase

$$S ::= [] \mid X \mid X \mathbin{@} X$$

$$\begin{aligned}
X ::= \ & \textbf{database}(db) \mid \textbf{yield } Y \mid \textbf{if } Z \textbf{ then yield } Y \\
\mid \ & \textbf{for } x \textbf{ in database}(db).f \textbf{ do } X
\end{aligned}$$

$$Y ::= x \mid \{\overline{f = Z}\}$$

$$Z ::= c \mid x.f \mid op(\overline{X}) \mid \textbf{exists } S$$

Figure 6: Normalized terms

2.3 Security Condition

The goal of the type system is to enforce a notion of noninterference for functional languages. Noninterference formalizes a notion of 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.

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

When the type is evident from the context, we omit the subscript on $\sim$. Moreover, we also write $\sim$ when referring to sequences of values. Base types are compared using ordinary equality if the values are considered public. In the case of function types

$$\frac{\ell = \mathtt{L} \Rightarrow k = k'}{k \sim_{\mathbf{int}^\ell} k'} \qquad \frac{\ell = \mathtt{L} \Rightarrow k = k'}{k \sim_{\mathbf{string}^\ell} k'} \qquad \frac{\ell = \mathtt{L} \Rightarrow k = k'}{k \sim_{\mathbf{bool}^\ell} k'}$$

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

$$\frac{\begin{array}{c}\forall v_1, v_2, v_1', v_2', \Omega_1, \Omega_2. \\ \Omega_1 \sim_\Sigma \Omega_2 \wedge v_1 \sim_t v_2 \wedge \\ e_1[f \mapsto \mathbf{rec}\ f(x) \to e_1, x \mapsto v_1] \longrightarrow^*_{\Omega_1} v_1' \wedge \\ e_2[f \mapsto \mathbf{rec}\ f(x) \to e_2, x \mapsto v_2] \longrightarrow^*_{\Omega_2} v_2' \Rightarrow \\ v_1' \sim_{t'} v_2'\end{array}}{\mathbf{rec}\ f(x) \to e_1 \sim_{t \to t'} \mathbf{rec}\ f(x) \to e_2} \qquad \frac{v_1 \sim_{t_1} v_1' \qquad v_2 \sim_{t_2} v_2'}{(v_1, v_2) \sim_{t_1 * t_2} (v_1', v_2')}$$

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

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

Figure 7: Introduction rules for $\sim_t$

and quoted expressions, $\sim_t$ corresponds to a notion of noninterference for the presented language.

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 $l_1 = \mathbf{yield}\ 1\ @\ []$ and $l_2 = \mathbf{yield}\ 2\ @\ []$. If the lists are typed with the type $t = (\mathbf{int}^{\mathtt{H}}\ \mathbf{list})^{\mathtt{L}}$, the length of the list is considered public, while the contents are confidential. If in contrast the type is $t' = (\mathbf{int}^{\mathtt{L}}\ \mathbf{list})^{\mathtt{L}}$, 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.

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 the baseline policy of *noninterference* [GM82a]. 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 expressions e_1 and e_2 are noninterfering 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 expresses by Σ and t, i.e., no secret parts of the database as defined by Σ are able to influence the public parts of the returned value as defined by t.

As common [Sys, MZZ+01, Sim03] in this setting, noninterference is *termination-insensitive* [VSI96, SM03b] 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.

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 cannot 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.

When lists are constructed using **yield** and [] they can be assigned an arbitrary level. **union** expressions 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.

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 an expression of type t (rule Run).

Expressions of the form **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 allow 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 list xs. We assume xs to have type $(t\ \mathbf{list})^{\ell}$ for some type t and level ℓ.

```
for x in xs do yield 1
```

Since the resulting list will have the same length as xs, the level of the result type must be at least ℓ. Moreover, the structure of the list produced for each element of xs is also revealed in the result. Assume that ys has type $(t'\ \mathbf{list})^{\ell}$ and consider the program:

```
for x in yield 1 do ys
```

Since this program evaluates to ys, the resulting type must also have at least level ℓ'. Hence, the typing rule requires the result type to be $(t'\ \mathbf{list})^{\ell \sqcup \ell'}$.

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

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

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

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

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

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

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

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

$$\textsc{If}\ \frac{\Gamma \vdash L : \mathbf{bool}^{\ell} \qquad \Gamma \vdash M : (t\ \mathbf{list})^{\ell'}}{\Gamma \vdash \mathbf{if}\ L\ \mathbf{then}\ M : (t\ \mathbf{list})^{\ell \sqcup \ell'}} \qquad \textsc{Run}\ \frac{\Gamma \vdash M : \mathbf{Expr}\langle t \rangle}{\Gamma \vdash \mathbf{run}\ M : t}$$

$$\textsc{Quote}\ \frac{\Gamma; \cdot \vdash M : t}{\Gamma \vdash \texttt{<@}\ M\ \texttt{@>} : \mathbf{Expr}\langle t \rangle} \qquad \textsc{Sub}\ \frac{\ell \leq \ell' \qquad \Gamma \vdash M : t^{\ell}}{\Gamma \vdash M : t^{\ell'}}$$

Figure 8: Type system for host language

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

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

$$\textsc{ApplyQ}\ \frac{\Gamma; \Delta \vdash e_1 : t \to t' \qquad \Gamma; \Delta \vdash e_2 : t}{\Gamma; \Delta \vdash e_1\ e_2 : t'} \qquad \textsc{OpQ}\ \frac{\Sigma(op) = \overline{t} \to t \qquad \overline{\Gamma; \Delta \vdash M : t^{\ell}}}{\Gamma; \Delta \vdash op(\overline{M}) : t^{\bigsqcup \ell_i}}$$

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

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

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

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

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

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

$$\textsc{DatabaseQ}\ \frac{\Sigma(db) = \{\overline{f : t}\}}{\Gamma; \Delta \vdash \mathbf{database}(db) : \{\overline{f : t}\}} \qquad \textsc{Antiquote}\ \frac{\Gamma \vdash e : \mathbf{Expr}\langle t \rangle}{\Gamma; \Delta \vdash \texttt{(\%}\ e\ \texttt{)} : t}$$

Figure 9: Typing rules for quoted language

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* $\Omega_1 \sim_\Sigma \Omega_2$ *and* $\Sigma \vdash e : t$*, 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. □

The main generalization necessary for proving this statement consists of quantifying over arbitrary contexts to yield a sufficiently strong induction hypothesis.

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* $v'_1 \sim_t v'_2$.

The proof of this lemma is presented in Section 6.

3 Implementation

Since F# contains an abundance of features not relevant to the current development we implement a type inference algorithm and compiler for 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 [CLW13].

The type inference algorithm proceeds by generating constraints between types of occurring expressions. The algorithm attempts to resolve these constraints using unification [DM82]. To allow writing flexible programs, the implementation supports polymorphism for both levels and types. 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 woman is older than the man.

```
// import statements omitted

[<Literal>]
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 w in (%db).People do
            for m in (%db).People do
            if (c.Her = w.Id) &&
```

```
                (c.Him = m.Id) &&
                (w.Age > m.Age) then
            yield { name = w.Name
                  ; diff = w.Age - m.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 [CLW13]. 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 `qquery` from the implementation accompanying [CLW13]. 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. [CLW13], 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 at the URL 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. 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. Constructors $C_1, \ldots, C_l$ have the form C_i **of** t_i where t_i is the argument to 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 following terms are added to the syntax of expression in the language:

$e ::= \ldots \mid C_i\ e \mid \textbf{match}\ e\ \textbf{with}\ C_1\ x_1 \to e_1\ ;\ \ldots\ ;\ C_l\ x_l \to e_l$

Semantics The semantics is extended with the following rules for evaluation of constructors and pattern matching. The set of values, evaluation, and quotation contexts is also expanded as shown below:

$$(\textbf{match}\ C_i\ v\ \textbf{with} \mid C_1\ x_1 \to e_1 \mid \ldots \mid C_l\ x_l \to e_l) \longrightarrow e_i[x_i \mapsto v]$$

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

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

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

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 $\mathit{upgrade}(t, \ell)$ which denotes upgrading the type t to have at least level ℓ in its outermost components. Figure 10 shows the rule schemas for algebraic data types.

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

$$\textsc{Match}\ \frac{\Gamma \vdash e : (\alpha\ T)^\ell \qquad \forall 1 \leq i \leq l.\Gamma, x_i : t_i \vdash e_i : t}{\Gamma \vdash (\textbf{match}\ e\ \textbf{with} \mid C_1\ x_1 \to e_1 \mid \ \ldots\ \mid C_l\ x_l \to e_l) : upgrade(t, \ell)}$$

Figure 10: Typing rule schemas for algebraic data types

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

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

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 is built with. 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.
- 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 larger due to algebraic data types.

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 support 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 pattern matching.

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

$$\frac{\Gamma \vdash e : {}'a\ \mathbf{MyList}^\ell \qquad \Gamma \vdash e_1 : t \qquad \Gamma, x : ({}'a, {}'a\ \mathbf{MyList}^\ell) \vdash e_2 : t}{\Gamma \vdash \mathbf{match}\ e\ \mathbf{with} \mid \mathbf{Nil} \rightarrow e_1 \mid \mathbf{Cons}\ x \rightarrow e_2 : \mathit{upgrade}(t,\ell)}$$

The generated rules match the intuitions given for the rest of the type system. Since **match** expressions reveal information about the results of the branches (which have type t) as well as the structure of the list (i.e. level ℓ) that the expression matches on, the level of the resulting list is $\mathit{upgrade}(t, \ell)$. 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:

$$\frac{}{\Gamma \vdash \mathbf{Leaf} : (\,'a\ \mathbf{BinTree})^{\ell}} \qquad \frac{\Gamma \vdash e : (((\,'a\ \mathbf{BinTree})^{\ell_1} * \ 'a) * (\,'a\ \mathbf{BinTree})^{\ell_2})}{\Gamma \vdash \mathbf{Node}\ e : (\,'a\ \mathbf{BinTree})^{\ell_1 \sqcup \ell_2}}$$

$$\frac{\Gamma \vdash e_1 : t \qquad \begin{array}{c}\Gamma \vdash e : (\,'a\ \mathbf{BinTree})^{\ell_1 \sqcup \ell_2} \\ \Gamma, x : (((\,'a\ \mathbf{BinTree})^{\ell_1} * \ 'a) * (\,'a\ \mathbf{BinTree})^{\ell_2}) \vdash e_2 : t\end{array}}{\Gamma \vdash \mathbf{match}\ e\ \mathbf{with} \mid \mathbf{Leaf} \to e_1 \mid \mathbf{Node}\ x \to e_2 : \mathit{upgrade}(t, \ell_1 \sqcup \ell_2)}$$

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 =
  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 11. The example data and database schema [ren14] are courtesy of `postgresqltutorial.com`[2]. The database contains information about approx. 16000 rentals, 600 customers, and 1000 movies.

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.

[2]Permission has been granted to use the E-R diagram as well as the database in this work.

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.

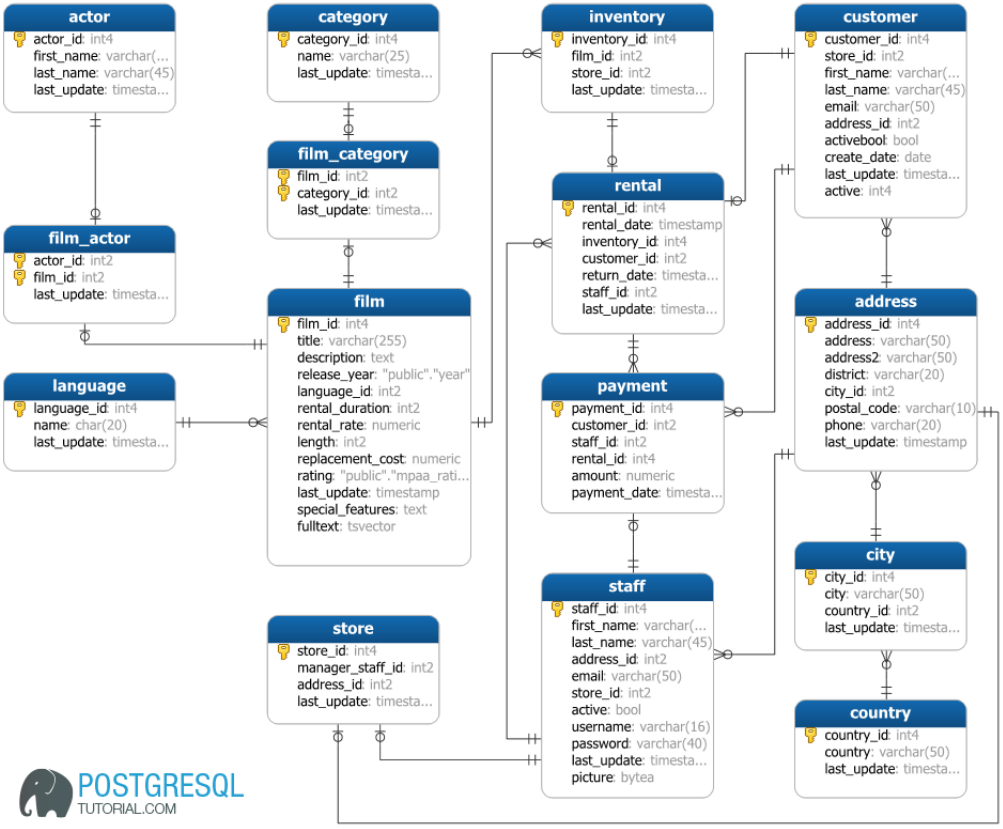

Figure 11: E-R diagram of movie rental database

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

The reason for the rightful type error is that first and last names of customers are typed as $\mathbf{string}^{\mathsf{H}}$ while the function `customersWhoRented` attempts to return a list containing elements of type $\mathbf{string}^{\mathsf{L}}$. 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 examples

above. 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 rather 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 might watch next. 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 might 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 might 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 it is not possible to generate 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.

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 Detailed soundness proof

This section presents a proof of Lemma 1 as well as its extension to algebraic data types.

Before proving Lemma 1, several auxiliary developments are needed.

Lemma 2. *Whenever* $\vdash v_1, v_2 : t^\ell$ *and* $\vdash v_1, v_2 : t^{\ell'}$ *where* $\ell \leq \ell'$*, and* $v_1 \sim_{t^\ell} v_2$*, then* $v_1 \sim_{t^{\ell'}} v_2$*.*

Proof. By induction over the structure of t.

If $\ell = \ell'$, there is nothing to show, so the only case that needs to be considered is $\ell = \mathtt{L}$ and $\ell' = \mathtt{H}$.

Base types: Immediate from $\sim_{t^\mathtt{H}}$ for base types, which always holds.

Function types, pair types, record types, $\mathbf{Expr}\langle t \rangle$: These types are not of the form t^ℓ, so there is nothing to prove.

List types: Since $\sim_{(t\ \mathbf{list})^\mathtt{H}}$ always holds, the statement follows immediately. □

Assumption 1. *We assume that all built-in operators preserve* $\sim$*:*

$$\forall op, \overline{v_1}, \overline{v_2}. \overline{v_1} \sim \overline{v_2} \Rightarrow \delta(op, \overline{v_1}) \sim_t \delta(op, \overline{v_2})$$

Lemma 3. *For all types* t*, we have that* $v_1 \sim_{upgrade(t,H)} v_2$ *for all values* $v_1, v_2 : t$*.*

Proof. The proof proceeds by induction over the type t:

Base types: Follows immediately from definition, since the resulting level annotation on the type will be `H` (e.g. $\mathbf{int}^\mathtt{H}$) and values of base types are always related by $\sim$ if the level annotation is `H`.

Function types: Follows immediately from the definition of $\sim$ for function types and induction hypothesis for result type.

Record types: By induction hypothesis we have that all fields of both records v_1 and v_2 are related by $\sim$. Then the conclusion follows by definition of $\sim$ for record types.

Pair types: Analogous to case for records.

List types: Immediate from definition of $\sim$ for lists, as $upgrade((t\ \mathbf{list})^\ell, \mathtt{H}) = (t\ \mathbf{list})^\mathtt{H}$, and $v_1 \sim_{(t\ \mathbf{list})^\mathtt{H}} v_2$ always holds.

Quoted expression type: By induction hypothesis for t. □

We can now prove Lemma 1. For clarity, we repeat the lemma statement here:

Lemma (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. Let $\overline{v_1} \sim \overline{v_2}$ and $\Omega_1 \sim_\Sigma \Omega_2$ and $e_1[\overline{x} \mapsto \overline{v_1}] \longrightarrow^*_{\Omega_1} v'_1$ and $e_2[\overline{x} \mapsto \overline{v_2}] \longrightarrow^*_{\Omega_2} v'_2$. We now have to prove that $v'_1 \sim_{t'} v'_2$:

We proceed by mutual induction on the typing derivation for the host and quoted language, where the proposition to prove for the quoted language is:

If $\Gamma; \overline{x : t} \vdash e : t$, $eval_{\Omega_1}(norm(e[\overline{x} \mapsto \overline{v_1}])) = v'_1$, $eval_{\Omega_2}(norm(e[\overline{x} \mapsto \overline{v_2}])) = v'_2$, $\Omega_1 \sim_\Sigma \Omega_2$ and $\overline{v_1} \sim \overline{v_2}$, then $v'_1 \sim_t v'_2$.

Case Const, $e = c$:

Constants are values and don't contain free variables or use databases. Therefore we have that $v'_1 = v'_2 = c$ and therefore $v'_1 \sim_{\Sigma(t)} v'_2$, since equal values are always low-equivalent.

Case Var, $e = y$:

Since e is well typed, y occurs in the typing context $\overline{x : t}$. Hence, y will be replaced with values $v_{1,i}$ (resp. $v_{2,i}$) from $\overline{v_1}$ (resp. $\overline{v_2}$), for some i. Since we assume $\overline{v_1} \sim \overline{v_2}$, we also have that $v'_1 = v_{1,i} \sim v_{2,i} = v'_2$, as desired.

Case Fun, $e = \mathbf{fun}(y) \to e' : (t \to t'^{\ell})^\ell$:

First note that in this case e is a value, so $v'_1 = v'_2 = e$. By induction hypothesis we have that whenever $e'[\overline{x} \mapsto \overline{v_i}] \to_{\Omega_i} v'_i$ (for some $i = 1{,}2$ and some $\overline{v_i}, v'_i$), then $v'_1 \sim_{t'} v'_2$. Hence the conclusion follows by the definition of $\sim$ for function types.

Case Rec, $e = \mathbf{rec}\ y(e') \to : (t \to t'^{\ell})^\ell$:

Analogous to case for Fun, except that the induction hypothesis is applied with a context that also includes y.

Case Apply, $e = f\ e' : t^\ell$:

By the premises of Apply we have that $e' : t'$ and $f : (t' \to t)$. By induction hypothesis we have that $e'[\overline{x} \mapsto \overline{v_1}]$ and $e'[\overline{x} \mapsto \overline{v_2}]$ evaluate to values related by $\sim_{t'}$. Also, by induction hypothesis we have that $f[\overline{x} \mapsto \overline{v_1}]$ and $f[\overline{x} \mapsto \overline{v_2}]$ evaluate to functions related by $\sim$. Hence by the definition of $\sim$ for functions, we conclude that if $(f\ e')[\overline{x} \mapsto \overline{v_i}] \to^*_{\Omega_i} v'_i$ (for $i = 1{,}2$), then $v'_1 \sim_{t^\ell} v'_2$.

Case Op, $e = op(\overline{M}) : t'^{\ell'}$:

In this case we have $\Sigma(op) = \bar{t} \to t'$ and $\overline{\Gamma \vdash M : t^\ell}$ with $\ell' = \bigsqcup \ell_i$. By induction hypothesis we get that all arguments evaluate to values related by $\sim$. Then the statement follows using Assumption 1.

Case Pair, $e = (e_1, e_2) : t_1 * t_2$:

By induction hypothesis we have that if $e_1[\overline{x} \mapsto \overline{v_i}] \longrightarrow_{\Omega_i} v'_i$ for $i = 1{,}2$ and $\overline{v_1} \sim \overline{v_2}$, then $v'_1 \sim_{t_1} v'_2$ and analogously for $e_2[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} w'_i$. In that case, we have that $e[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} (v'_i, w'_i)$. By definition of $\sim_{t_1 * t_2}$ it follows from $v'_1 \sim_{t_1} v'_2$ and $w'_1 \sim_{t_2} w'_2$, that $(v'_1, w'_1) \sim_{t_1 * t_2} (v'_2, w'_2)$ as desired.

Case Fst, $e = \mathbf{fst}\ e' : t'_1$ where $\overline{x : t} \vdash e' : t'_1 * t'_2$:

By induction hypothesis , if $e'[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} v'_i$, then $v'_i = (w'_i, z'_i)$ (for $i = 1{,}2$) and $(w'_1, z'_1) \sim_{t'_1 * t'_2} (w'_2, z'_2)$. By definition of $\sim_{t'_1 * t'_2}$, this entails that $w'_1 \sim_{t'_1} w'_2$, which are the evaluation results for $e[\overline{x} \mapsto \overline{v_i}]$.

Case SND, $e = \mathbf{snd}\ e' : t'_2$ where $\overline{x : t} \vdash e' : t'_1 * t'_2$:

Analogous to case FST.

Case RECORD, $e = \{\overline{f = M}\}$:

By induction hypothesis we have that for every M_i in $\overline{M}$, it holds that if $M_i[\overline{x} \mapsto \overline{v_j}] \longrightarrow^*_{\Omega_j} v'_j$ (for j = 1,2), then $v'_1 \sim v'_2$. Since $\sim$ for record types relates records with the same fields containing related values, this concludes the case.

Case PROJECT, $e = e'.f$:

By induction hypothesis , if $e'[\overline{x} \mapsto w_i] \longrightarrow^*_{\Omega_i} w'_i$ (for i = 1,2), then $w'_1 \sim w'_2$. Since e' is typed as a record, we have that the fields of w'_1 and w'_2 are related by $\sim$. Since f is one of those fields, the results of evaluating $e[\overline{x} \mapsto v_i]$ for i = 1,2 are also related by $\sim$, concluding the case.

Case YIELD, $e = \mathbf{yield}\ e' : (t'\ \mathbf{list})^\ell$ and $e' : t$:

By induction hypothesis we have that $e'[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} v'_i \Rightarrow v'_1 \sim_{t'} v'_2$ (for i = 1,2. Hence the desired conclusion follows by the definition of $\sim_{(t'\ \mathbf{list})^\ell}$: If $\ell = \mathtt{H}$, there is nothing to prove. Otherwise, the lists contain one element each and their elements are related by $\sim$.

Case NIL, $e = []$: Immediate.

Case UNION, $e = M@N : (t'\ \mathbf{list})^{\ell \sqcup \ell'}$ where $M : (t'\ \mathbf{list})^\ell$ and $N : (t'\ \mathbf{list})^{\ell'}$:

By induction hypothesis we get that $M[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} v'_i \Rightarrow v'_1 \sim_{t'\ \mathbf{list}} v'_2$ and $N[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} v'_i \Rightarrow v'_i$ for $i = 1, 2$ and any v'_i. If either $\ell = \mathtt{H}$ or $\ell' = \mathtt{H}$, there is nothing to show, since in this case $\ell \sqcup \ell' = \mathtt{H}$.

If $\ell, \ell' = \mathtt{L}$, we have that the resulting lists for both M and N must have the same lengths and the elements are related by $\sim_t$. Hence, the concatenation of these lists is related by $\sim_{(t'\ \mathbf{list})^{\ell \sqcup \ell'}}$.

Case FOR, $e = \mathbf{for}\ y\ \mathbf{in}\ M\ \mathbf{do}\ N : (t'\ \mathbf{list})^{\ell \sqcup \ell'}$ where $M : (t''\ \mathbf{list})^\ell$ and $N : (t'\ \mathbf{list})^{\ell'}$:

Again we only need to consider $\ell = \mathtt{L} \wedge \ell' = \mathtt{L}$, since the goal is trivial otherwise.

By induction hypothesis we have that $M[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} v'_i \Rightarrow v'_1 \sim_{(t''\ \mathbf{list})^\ell} v'_2$. Since e evaluates to a value, we have $M[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} w_i$ for some w_1, w_2, where $|w_1| = |w_2|$ since $\ell = \mathtt{L}$.

We then show the main goal by induction over the structure of w_1 and w_2:

Case $w_1 = w_2 = []$: The conclusion follows immediately, since the entire **for**-expression then evaluates to [] regardless of N.

Case $w_1 = [w'_1]$ and $w_2 = [w'_2]$: In this case, e evaluates to the result of $N[\overline{x} \mapsto \overline{v_i}][y \mapsto w'_i]$ for i = 1,2.

By the induction hypothesis we also obtain that whenever $N[\overline{x} \mapsto \overline{v_i}][y \mapsto w'_i] \rightarrow^*_{\Omega_i} v'_i$, then $v'_1 \sim_{(t'\ \mathbf{list})^{\ell'}} v'_2$, which concludes this case.

Case $w_1 = w'_1 @ w''_1$ and $w_2 = w'_2 @ w''_2$: Follows by induction hypotheses for the two sub-lists.

Case EXISTS, $e = \mathbf{exists}\ M : \mathbf{bool}^\ell$ where $M : (t\ \mathbf{list})^\ell$: If $\ell = \mathtt{H}$, we are done. Otherwise we have from the induction hypothesis that $M[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} v'_i \Rightarrow v'_1 \sim_{(t\ \mathbf{list})^\ell} v_2$. Hence, the resulting lists must have the same number of elements, which entails the same result for the entire expression.

Case IF, $e = \mathbf{if}\ M\ \mathbf{then}\ N : (t'\ \mathbf{list}^{\ell \sqcup \ell'})$ where $M : (t''\ \mathbf{list})^\ell)$ and $N : (t'\ \mathbf{list})^{\ell'}$:

By induction hypothesis we have that $M[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} v'_i \Rightarrow v'_1 \sim_{(t'' \textbf{ list})^\ell} v'_2$. Again, the goal is trivial if $\ell = \texttt{H}$ or $\ell' = \texttt{H}$.

Otherwise, we have evaluating $M[\overline{x} \mapsto \overline{v_i}]$ results in the same boolean for $i = 1{,}2$, and hence we know from the induction hypothesis that evaluation of $N[\overline{x} \mapsto \overline{v_i}]$ also yields related results.

Case RUN, $e = \textbf{run}\ e' : t$ where $e' : \textbf{Expr}\langle t \rangle$: Follows by induction hypothesis for the quoted language.

Case LIFT, $e = \textbf{lift}\ e'$: Immediate for the induction hypothesis for e'.

Case SUB, $\ell \leq \ell'$: See lemma 2.

The cases for the typing rules for the same constructs in the quoted language, are practically identical to the previous cases. We now prove the cases specific to the type system for the quoted language:

Case ANTIQUOTE, $e = \texttt{(\% } e' \texttt{)}$: Follows trivially from induction hypothesis .

Case DATABASE, $e = \textbf{database}(db)$: Follows from the assumption that $\Omega_1 \sim_\Sigma \Omega_2$, since evaluation of e results in $\Omega_i(db)$ for $i = 1{,}2$.

Extension to algebraic data types: We now prove the cases for algebraic data types in the language. Again we assume that the algebraic data type is of the form $\alpha\ T = C_1\ t_1 | \ldots | C_l\ t_l$ with α being a (possibly empty) product of type variables, which can occur in $t_1, \ldots, t_l$.

Case CONSTR, $e = C_i\ e'$ where $e : \alpha\ T^{\bigsqcup_{T^\ell \in t_i} \ell}$:

Let $\ell' = \bigsqcup_{T^\ell \in t_i} \ell$. If $\ell' = \texttt{H}$, there is nothing to prove.

Let $\ell' = \texttt{L}$. According to the semantics for constructor applications, we have that $v'_1 = C_i\ v''_1$ and $v'_2 = C_i\ v''_2$, where $e'[\overline{x} \mapsto \overline{v_i}] \longrightarrow^*_{\Omega_i} v''_i$. By the induction hypothesis for the typing derivation of e' we have that $v''_1 \sim_{t_i} v''_2$. By definition of $\sim$ for constructors, this entails that $v'_1 \sim v'_2$.

Case MATCH, $e = \textbf{match}\ e'\ \textbf{with}\ C_i\ x_i \to e_i$:

Let $\overline{x : t} \vdash e : t'^{\ell \sqcup \ell'}$, $\overline{x : t} \vdash e_i : t'^{\ell'}$ for $i = 1, \ldots, l$, and $e' : \alpha\ T^\ell$. If $\ell = \texttt{H}$, we conclude with Lemma 3. If $\ell = \texttt{L}$ and $e'[\overline{x} \mapsto \overline{v_j}] \longrightarrow^*_{\Omega_i} v''_j$ for $j = 1{,}2$, then by induction hypothesis for the typing derivation of e' and definition of $\sim$ for constructors, we have that $v''_j = C_k\ v'''_j$ for some k where $1 \leq k \leq l$. Therefore, the same branch e_k will be executed for $e[\overline{x} \mapsto \overline{v_1}]$ and $e[\overline{x} \mapsto \overline{v_2}]$. By induction hypothesis for the typing derivation of e_k, we have again that if $e_k[\overline{x} \mapsto \overline{v_j}] \longrightarrow^*_{\Omega_i} w'_j$, then $w'_1 \sim_{t'^{\ell'}} w'_2$, which concludes this case, as $e[\overline{x} \mapsto \overline{x_j}] \longrightarrow^*_{\Omega_i} w'_j$.

□

7 Related Work

Until recently, little work has been on bridging information-flow controls for applications [SM03b, Gue07, HS11, Bie13] and databases they manipulate. While mainstream database management systems such as PostgreSQL [Pos14], SQLSever [SQL14], and MySQL [MyS14] include protection 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. [HYH+04] 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 [LZ05b] 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 [SS09]. 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. [CLWY06], 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. [YYW+07] 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 [BG09] 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. [CSH09] also builds on Links. With the Fable type system by Swamy et al. [SCH08] 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 [DC10] 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 [Chl10] 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. [CPS+11] 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 [LC13]. 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. [GLS+12] 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 [LC13] update, delete, and privilege delegation.

IFDB by Schultz and Liskov [SL13] 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. [CPIA13] 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 [HR98] 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 [PS02], which they have developed for a core of ML, and which serves as the base for the Flow Caml tool [Sim03]. 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 [PS02].

The tools like SIF [CVM07], SWIFT [CLM$^+$09], and Fabric [LGV$^+$09] allow the programmer to enforce powerful policies for confidentiality and integrity in web applications. The programmer labels data resources in the source program with fine-grained policies using Jif [MZZ$^+$01], 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 [NS08] 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 [imd14].

8 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 back-end). 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 end-to-end security.

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 [SS09] 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* [SS01, LZ05a, LZ05b] is released, where we can express aggregates through *escape hatches* [SM03a], 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* [DF02] 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.

References

[BG09] Ioannis G. Baltopoulos and Andrew D. Gordon. Secure compilation of a multi-tier web language. In *TLDI*, pages 27–38, 2009.

[Bie13] Nataliia Bielova. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *J. Log. Algebr. Program.*, pages 243–262, 2013.

[BRS10] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Unifying Facets of Information Integrity. In *ICISS*, pages 48–65, 2010.

[Chl10] Adam Chlipala. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *OSDI*, pages 105–118, 2010.

[CLM+09] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Building secure web applications with automatic partitioning. *Commun. ACM*, 52(2):79–87, 2009.

[CLW13] James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *ICFP*, pages 403–416. ACM, 2013.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *FMCO*, pages 266–296, 2006.

[CPIA13] Georgios Chinis, Polyvios Pratikakis, Sotiris Ioannidis, and Elias Athanasopoulos. Practical information flow for legacy web applications. In *ICOOOLPS*, pages 17–28, 2013.

[CPS+11] Luís Caires, Jorge A. Pérez, João Costa Seco, Hugo Torres Vieira, and Lúcio Ferrão. Type-Based Access Control in Data-Centric Systems. In *ESOP*, pages 136–155, 2011.

[CSH09] Brian J. Corcoran, Nikhil Swamy, and Michael W. Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD Conference*, pages 269–282, 2009.

[Cur34] H. Curry. Functionality in combinatorial logic. In *Proceedings of National Academy of Sciences*, volume 20, pages 584–590, 1934.

[CVM07] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proc. USENIX Security Symposium*, pages 1–16, August 2007.

[DC10] Benjamin Davis and Hao Chen. DBTaint: Cross-application Information Flow Tracking via Databases. In *WebApps*, pages 12–12. USENIX Association, 2010.

[DF02] Josep Domingo-Ferrer, editor. *Inference Control in Statistical Databases, From Theory to Practice*, volume 2316 of *LNCS*. Springer, 2002.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212. ACM, 1982.

[GLS+12] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *OSDI*, pages 47–60, 2012.

[GM82a] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. IEEE SP*, pages 11–20, April 1982.

[GM82b] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[Gue07] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[GWT14] Google Web Toolkit. `http://www.gwtproject.org/`, 2014. Accessed: 2014-02-20.

[HF92] Paul Hudak and Joseph H. Fasel. A Gentle Introduction to Haskell. *SIGPLAN Notices*, 27(5):1–, 1992.

[How] William Howard. The formulae-as-types notion of construction. pages 479–490.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *POPL*, pages 365–377, 1998.

[HS11] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. *Proc. of the 2011 Marktoberdorf Summer School. IOS Press*, 2011.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, pages 40–52, 2004.

[imd14] Internet Movie Database. `http://www.imdb.com/`, 2014. Accessed: 2014-02-20.

[LC13] Luísa Lourenço and Luís Caires. Information Flow Analysis for Valued-Indexed Data Security Compartments. In *TGC*, 2013.

[LGV+09] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: a platform for secure distributed computation and storage. In *SOSP*, pages 321–334, 2009.

[LIN14] LINQ (Language-Integrated Query). `http://msdn.microsoft.com/en-us/library/bb397926.aspx`, 2014. Accessed: 2014-02-20.

[LZ05a] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, pages 158–170, 2005.

[LZ05b] Peng Li and Steve Zdancewic. Practical Information-flow Control in Web-Based Information Systems. In *CSFW*, 2005.

[MSS11] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and Guarantees for Compositional Noninterference. In *CSF*, pages 218–232. IEEE Computer Society, 2011.

[MyS14] Privileges Provided by MySQL. `https://dev.mysql.com/doc/refman/5.1/en/privileges-provided.html`, 2014. Accessed: 2014-02-20.

[MZZ+01] 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.

[NS08] Arvind Narayanan and Vitaly Shmatikov. Robust De-anonymization of Large Sparse Datasets. In *IEEE Symp. on Security and Privacy*, 2008.

[OWA13] 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.

[Pie02] Benjamin Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[Pos14] Database Roles and Privileges. `http://www.postgresql.org/docs/9.0/static/user-manag.html`, 2014. Accessed: 2014-02-20.

[PS02] François Pottier and Vincent Simonet. Information flow inference for ML. In *POPL*, pages 319–330. ACM, 2002.

[ren14] PostgreSQL sample database. `http://www.postgresqltutorial.com/postgresql-sample-database/`, 2014. Accessed: 2014-02-20.

[rub14] Ruby on Rails. `http://rubyonrails.org/`, 2014. Accessed: 2014-02-20.

[SCH08] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A Language for Enforcing User-defined Security Policies. In *IEEE Symp. on Security and Privacy*, 2008.

[Sim03] Vincent Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, 2003.

[SL13] David A. Schultz and Barbara Liskov. IFDB: decentralized information flow control for databases. In *EuroSys*, pages 43–56, 2013.

[SM03a] A. Sabelfeld and A. C. Myers. A Model for Delimited Information Release. In *ISSS*, volume 3233 of *LNCS*, pages 174–191, 2003.

[SM03b] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, pages 5–19, 2003.

[SQL14] Authorization and Permissions in SQL Server. `http://msdn.microsoft.com/en-us/library/bb669084(v=vs.110).aspx`, 2014. Accessed: 2014-02-20.

[SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Trans. Comput. Syst.*, pages 277–288, 1984.

[SS01] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *Higher Order and Symbolic Computation*, 14(1):59–91, March 2001.

[SS09] A. Sabelfeld and D. Sands. Declassification: Dimensions and Principles. *J. Computer Security*, 17(5):517–548, January 2009.

[Sym06] Don Syme. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution. In *Workshop on ML*, pages 43–54. ACM, 2006.

[Sys] Praxis High Integrity Systems. SPARKAda Examinar. Software release. `http://www.praxis-his.com/sparkada/`.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *J. Computer Security*, 4(3):167–187, 1996.

[YYW+07] Sachiko Yoshihama, Takeo Yoshizawa, Yuji Watanabe, Michiharu Kudo, and Kazuko Oyanagi. Dynamic Information Flow Control Architecture for Web Applications. In *ESORICS*, pages 267–282, 2007.