Type-based Access Control in Data-Centric Systems

Luís Caires[1], Jorge A. Pérez[1], João C. Seco[1], Hugo T. Vieira[1], and Lúcio Ferrão[2]

[1] CITI and Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
[2] OutSystems SA

Data-centric multi-user systems, such as web applications, require flexible yet fine-grained data security mechanisms. Such mechanisms are usually enforced by a specially crafted security layer, which adds extra complexity and often leads to error prone coding, easily causing severe security breaches. In this paper, we introduce a programming language approach for enforcing access control policies to data in data-centric programs by static typing. Our development is based on the general concept of refinement type, but extended so as to address realistic and challenging scenarios of permission-based data security, in which policies dynamically depend on the database state, and flexible combinations of column- and row-level protection of data are necessary. We state and prove soundness and safety of our type system, stating that well-typed programs never break the declared data access control policies.

1 Introduction

Data-centric multi-user software systems are a pervasive class of software applications, where transactions manipulate information stored in a shared database on behalf of several different users, playing several different roles. In the case of web-based systems, of which common examples are collaborative applications or social networks, the number of users may be extremely large, and the security requirements critical. Indeed, such systems require very flexible yet fine-grained data security mechanisms, including dynamic, role-based access control. Moreover, web applications are usually developed and executed in heterogeneous multiple-tier environments. Access control to data in such environments is typically performed at runtime by specially crafted security code, which mediates between the application code and the relational database management system. Such a security layer is hard to construct, error prone, and may easily cause severe security breaches. To make things a bit harder, access control policies are usually dependent on stored data and meta-data, and highly dynamic. Addressing such security requirements is frequently hindered by the expressiveness gap that exists between the required access control policies at the application side, and the actual security mechanisms provided by database engines.

Properly mapping the access control policies defined at the application side into associated database mechanisms is often difficult, if not impossible, also because multiple application profiles should be related to only a few database profiles. As an unfortunate side result, the enforcement of access control policies at the database level is kept to a minimum, promoting security breaches, as a consequence of the lack of protection between layers. It is therefore important to identify new verification methods to prevent programmers from inadvertently violating access control constraints in such common scenarios of permission-based data-centric security.

In this work we develop a programming language approach for expressing and verifying access control policies to data in (relational) data-centric programs by means of static type checking. More precisely, we introduce a core language λ_{DB} which includes typeful programming constructs to manipulate (query and update) data stored in data "entities", to be physically represented by database tables (cf. the relational model). The associated type system allows access control policies to be associated to data entities, allowing queries and updates to the database to be validated against the declared constraints, taking into account also the particular information stored, and the current static state of the current principal. In λ_{DB}, access control policies are explicitly represented at the level of types: we endow λ_{DB} with dependent refinement types, which ensure that well-typed programs do not violate prescribed access control policies.

Access control mechanisms available in database systems are supported by fixed relations between users, operations, and tables. This basic approach induces a so called "column-level" protection in database tables, based on the *static structure* of the data model. Such a static form of access control, however, is far from enough, because in common situations the authorization to access a particular piece of information depends on information also stored in the database. For example, in web applications, it is very common to find data with security requirements such as, e.g., "only the manager of a proposal is able to modify it" or "only intranet users can see the submitted applications". Similarly, dynamic properties such as "only the current friends of a user can see his/her photos" are familiar. Notice that, the predicate "friend" as used here is dynamic and state dependent: a user can be granted a permission (by being selected as "friend"), that may be later revoked by other transaction. Any access control mechanism for data-centric systems should therefore be flexible and expressive enough to capture state dependencies of these kinds, which covers the notion of "row-level" protection to database entities. Our type system fully addresses such a challenging combination of column-level, row-level, and authorization permissions to enable the static verification of policies such as the one above, enabling the type-checker to issue an error whenever the programmer inadvertently tries to compile insecure code.

Our approach is based on associating expressive conditions as guards to basic operations on data entities (read, update, insert, etc.), and verify such conditions at the appropriate points in the code by static checking. In this way, it is possible to encode usual database permissions (but also more general conditions), and verify the program's conformance to the access control policies. To this end we build on the notion of *refinement types* [13, 15], extended to a setting where properties depend both on the static state and on the dynamic state when manipulating entity tables. Intuitively, a refinement type $\{x : \beta \mid C\}$ classifies values of type β for which the logical expression C holds; for instance, $\{x : \mathsf{int} \mid x > 0\}$ is the type of the positive integers. This also allows a type environment to be seen as a refined property of the declared objects, so that for instance, we may say that a typing environment $\Delta = \Delta', a : \{x : \beta \mid C\}$ entails $C\{a/x\}$, written $\Delta \vdash C\{a/x\}$, where C is some condition about the program state. We also consider the combination of refinement types with functional dependent types, something particularly useful to express general pre- and post- conditions [15]. In the context of access control to database entities, refinement types are useful to express what conditions are valid for each program fragment and, ultimately, to implement flex-

```
entity Person [ userid: string; public: string;
                photo: picture; secret: string; ]
  read public where true;
  read userid where Auth(uid);
  read secret where Auth(uid) and uid = userid;
  read photo where Auth(uid) and Friends(userid,uid);
  write where Auth(userid);

entity Friend [ user: string; friend: string; ]
  invariant Friends(user,friend)
  write where Auth(user)
```

Fig. 1. Some Sample Database Entities.

ible mechanisms for access control. Notice however, that our requirements for dynamic row-level protection, as captured in the typing rules for database reading and updating constructs, are not naturally expressible in existing refinement type systems, due to the dependence of refinements on the actual data stored in database entities.

We illustrate our approach with a sequence of simple examples, in the scenario of a social network application. The data model contains, among other elements, entities *Person* and *Friend*, defined in Fig. 1. Entities are defined by enumerating their field names and types, together with a set of access control policies associated to them. Here we focus on the "friendship" relation, implemented by entity `Friend`, and on using it to control the access users have to each others' data. Each permission clause is composed by (i) the kind of access granted (either read or write); (ii) the list of entity fields it protects; and (iii) a condition, expressed as a logical formula. Field names occur in the logical conditions, to allow them to refer to data in the entity row. The intended semantics is that the disjunction of the set of all access control policies for a given kind of access and field name must be valid for the corresponding operations to be applied. Intuitively, the capabilities associated to a list of database columns will only be granted if the associated conditions hold in the context of evaluation. In general, the evaluation context entails primitive properties (e.g., expressing authorizations), which may be explicitly asserted, or hold as a consequence of logical deduction from the primitive ones (conceptually stored in a *log* [15]). The elementary propositions of such formulas are predicates parameterized by language identifiers and constants.

In our case, entity `Person` declares four read permissions and one write permission. The first read permission stipulates that field `public` is always readable, as its associated condition (true) always holds. The contents of field `userid` of a row is only readable if `Auth(uid)` holds for some `uid` (identifiers not field names are existentially quantified variables). The third read permission states that field `secret` should be accessible only if predicate `Auth(userid)` holds in the current state. When a condition in a permission clause refers to field names, its validity depends on the actual data stored on each entity row. Free names in the conditions (for example `uid`) are existential parameters in each permission, and will be instantiated by concrete values at verification time. In the example, we assume that `Auth(`$user_1$`)` asserts that user named $user_1$ is authenticated in the system. The consequence of this condition is that an au-

thenticated user can only select the field `secret` from those rows in which field `userid` corresponds to its own userid (for brevity, we omit the (trusted) code that establishes predicate `Auth(_)` in the login process). The fourth read permission states that an authenticated user (with name $user_2$) can access the field `photo` of another user (with name $user_1$) only in the case that predicate `Friends(`$user_1$`,`$user_2$`)` is valid. The write permission applies to every field in `Person` and is self-explanatory.

The specification of entity *Friend* features a write permission and an invariant clause. The invariant clause says that for each one of the rows $[user = user_1, friend = user_2]$ actually stored in entity `Friend`, the proposition `Friends(`$user_1$`,`$user_2$`)` always holds. Intuitively, the invariant must hold of a tuple in order for it to be added to the table, so that it is known to hold for every tuple read from the table; invariants in entity declarations express refinements over the actually stored data. As specified by the write permission, a row such as $[user = user_1, friend = user_2]$, asserting $user_2$ to be a friend of $user_1$ can only be created, updated, or deleted in a context where the condition (authorization) `Auth(`$user_1$`)` holds. The permissions thus enforce that friends can only be added (or removed) by the authenticated user $user_1$ to a data record having her as key, and only if the condition `Friends(`$user_1$`,`$user_2$`)` holds in the current state. This last condition may hold either because it may be obtained as effect of a query, or by calling a trusted library (which may establish it by an explicit assume statement).

Permission clauses as introduced in this example, also support reasoning about row- and column-level protection when accessing data. For instance, the following query, expressed in a LINQ [16] like syntax,

```
from p in Person where true select public
```

is well-typed and safe since the condition in the permission associated to field `public` always holds (true). Other fields are protected by stronger conditions. For instance, reading the contents of the `secret` field of a row is only possible for the rows where `userid=`*name* and `Auth(`*name*`)`. For example, the query

```
from p in Person where p.userid=loggeduser select secret
```

will only type-check in a context where `Auth(loggeduser)` holds, for some given `loggeduser` while the query

```
from p in Person where true select secret
```

will not type-check. We now consider the definitions of some functions on entities. Consider a function that fetches the field `secret` of a given user, with type

```
getSecret: {n:string | Auth(n)} → string
```

The refined parameter type acts as a pre-condition, since function `getSecret` can only be called if the argument is a string `s` such that predicate `Auth(s)` holds. We define `getSecret` using a query on entity `Person`, as follows:

```
def getSecret(name:{n:string | Auth(n)}):string = {
  let l = (from p in Person
           where p.userid = name
           select p.secret) in
  if isEmpty(l) then NULL else head(l) }
```

According to the declarations in Fig. 1, the context of the query expression selecting field `secret` is required to satisfy predicate `Auth(p.userid)` for the selected rows. This is obtained directly from the typing `name:{n:string | Auth(n)}` and from the query's where clause `p.userid = name`, which holds for all rows in the result of the query. Notice that we assumed given two predefined operations `isEmpty` and `head` to handle the results of queries.

As a last example, we define a function for fetching the photo of a given user. We will need two parameters, the logged-on user name and the user name whose photo is sought. The function `getPhoto` would then have the following type:

```
getPhoto: {n:string | Auth(n)} × string → picture
```

Reading the field photo requires the owner of the photo to be known as a friend of the authenticated user, expressed by the predicate `Friends(−,−)`.

Recall that `Friends(`$user_i$`,`$user_j$`)` relation is managed dynamically by the application, by inserting and deleting rows from entity `Friend`. Now, given users $user_1$ and $user_2$ where we know that `Auth(`$user_1$`)` is valid (from the function type parameters), we may statically establish that predicate `Friends(`$user_2$`,`$user_1$`)` is valid. This is done by querying such a row in entity `Friend`. We thus have

```
def getPhoto(name:{n:string | Auth(n)},
             othername:string):picture = {
  if isFriendOf(othername,name) then
    let l = (from p in Person
             where p.userid = othername
             select p.photo) in
    if isEmpty(l) then NULL else head(l)
  else NULL }
```

The conditions for reading field `photo` can be deduced from the context, which entails `Auth(name)` (from the parameter type) and `Friends(othername,name)` (from the result of the `isFriendOf` call). Notice how our type system actually forces the programmer to perform a runtime test, by consulting the `Friend` entity before retrieving the photo. To that end, we may use an auxiliary function `isFriendOf`, with type

```
isFriendOf: u:string×v:string → {b:bool | b ⇒ Friends(u,v)}
```

and which encapsulates the table access, and whose return (refined) type entails the friendship of the given parameters, defined as follows:

```
def isFriendOf(user:string, friend:string):
      {b:bool | b ⇒ Friends(user,friend)} = {
  let l = (from f in Friends
           where f.user = user
                 and f.friend = friend
           select f) in
  if isEmpty(l) then false
  else {head(l); true} }
```

We assume that read permissions for `user` and `friend` are true by default. Notice that the type of `head(l)` entails `Friends(user,friend)`, resulting from the entity invariant.

$$
\begin{array}{lll}
e ::= & & \textit{(Expressions)} \\
 & v & \text{(Value)} \\
\mid & e\ (v) & \text{(Application)} \\
\mid & [\overline{m = e}] & \text{(Record)} \\
\mid & e.m & \text{(Field Selection)} \\
\mid & e\ op\ e & \text{(Operation)} \\
\mid & e\,?\,e:e & \text{(Conditional)} \\
\mid & \mathsf{let}\ x = e\ \mathsf{in}\ e & \text{(Let)} \\
\mid & e_1, ..., e_k & \text{(Collection)} \\
\mid & \mathsf{create}\ t : \beta_{\overline{\rho}}\ \mathsf{in}\ e & \text{(Create)} \\
\mid & \mathsf{from}\ x\ \mathsf{in}\ t\ \mathsf{where}\ e\ \mathsf{select}\ e & \text{(Select)} \\
\mid & \mathsf{update}\ x\ \mathsf{in}\ t\ \mathsf{where}\ e\ \mathsf{with}\ e & \text{(Update)} \\
\mid & \mathsf{append}\ e\ \mathsf{to}\ t & \text{(Append)} \\
\mid & \mathsf{delete}\ x\ \mathsf{in}\ t\ \mathsf{where}\ e & \text{(Delete)} \\
\mid & \mathsf{assume}\ C & \text{(Assume)} \\
\mid & \mathsf{assert}\ C & \text{(Assert)}
\end{array}
$$

$$
\begin{array}{lll}
C, R, W ::= & & \textit{(Propositions)} \\
 & p(\overline{V}) & \text{(Predicate)} \\
\mid & V = V & \text{(Equality)} \\
\mid & C \wedge C & \text{(Conjunction)} \\
\mid & C \implies C & \text{(Implication)}
\end{array}
$$

$$
\begin{array}{lll}
u, v ::= & & \textit{(Values)} \\
 & () & \text{(Unit value)} \\
\mid & \mathsf{true} & \text{(True)} \\
\mid & \mathsf{false} & \text{(False)} \\
\mid & x & \text{(Variable)} \\
\mid & \lambda x : \tau.e & \text{(Abstraction)} \\
\mid & [\overline{m = v}] & \text{(Record)} \\
\mid & v_1, ..., v_k & \text{(Collection)} \\
\mid & \star(v) & \text{(Classified Value)}
\end{array}
$$

$$
\begin{array}{lll}
V ::= & & \textit{(Terms)} \\
 & () & \text{(Unit value)} \\
\mid & \mathsf{true} & \text{(True)} \\
\mid & \mathsf{false} & \text{(False)} \\
\mid & x & \text{(Variable)} \\
\mid & \lambda x : \tau.e & \text{(Abstraction)} \\
\mid & [\overline{m = V}] & \text{(Record)} \\
\mid & V_1, ..., V_k & \text{(Collection)} \\
\mid & \star(V) & \text{(Classified Term)} \\
\mid & V.m & \text{(Field Selection)}
\end{array}
$$

$$
\begin{array}{lll}
\rho ::= & & \textit{(Permissions)} \\
 & \mathtt{rd}(m, R) & \text{(Read)} \\
\mid & \mathtt{wr}(m, W) & \text{(Write)}
\end{array}
$$

Fig. 2. Syntax of λ_{DB}: Expressions, Logical Propositions, Values, Terms, Permissions.

The rest of the paper is structured as follows. Sections 2 and 3 introduce the syntax and operational semantics of the λ_{DB} language, where a key notion of access control compliance is also defined (Definition 3.7). In Section 4, a type system for ensuring safety is proposed and its main results are stated, namely type preservation under reduction (Theorem 4.6), progress (Theorem 4.7), and typeful access control compliance (Corollary 4.8). Section 5 discusses related work.

2 Syntax

In this section we present the syntax of λ_{DB} and describe its main constructs. λ_{DB} contains a *functional core*, several constructs for *storage and manipulation* of data entities, and *logical operators* for expressing the knowledge about properties of data in programs. The syntax of λ_{DB} expressions (e), logical propositions (C), values (v), terms (V) and permissions (ρ) is given in Fig. 2, where we assume given infinite sets of names Λ (ranged over by $m, n, o, \ldots$) and of variables $\mathcal{V}$ (ranged over by $t, x, y, z, \ldots$). The distinguished variable *this* is used in table permissions to refer to a table row.

Expressions include values v, application $e(v)$, records $[m_1 = e_1, \ldots, m_k = e_k]$, where each m_i is a field, and field access $e.m$. λ_{DB} values include the unit value $()$, true and false, variables, and abstractions $\lambda x{:}\tau.e$, where τ is the type of x. A value may also be a record or a collection $v_1, \ldots, v_k$. We often use the overbar to abbreviate in-

dexed sets; this way, e.g., $[\overline{m = v}]$ stands for $[m_1 = v_1, \ldots, m_k = v_k]$ and $\overline{v}$ stands for $v_1, \ldots, v_k$. If $r' = [\ldots, m_i = v'_i, \ldots]$, then we denote v'_i by r'_{m_i}. Database tables are modeled by references to collections of records. To represent high security data (data not accessible according to the permissions and the current knowledge) we introduce *classified* values $\star(v)$, meaning that value v is not accessible to the current program. Classified values $\star(v)$ cannot appear in source programs, but are useful for expressing the language semantics and its properties, namely the notion of access control compliance (Definition 3.7). Notice that the language of values is included in the language of terms, and that the language of terms is included in the language of expressions.

A field access expression $e.m$, provided e evaluates to a record with a field named m, evaluates to the contents of the m field of such record. We assume some unspecified set of operations over basic values (ranged over by op) and write $e_1 \ op \ e_2$ to represent the application of the operation to the result of evaluating expressions e_1 and e_2. Expressions also include a conditional statement $e_1 \,?\, e_2 \,:\, e_3$, with the expected meaning: if e_1 then evaluate e_2 otherwise evaluate e_3. The let expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ assigns the value obtained by evaluating e_1 to variable x, and binds x with scope e_2 (notice that let is not representable through function application since we require the argument of application to be a value, for typing purposes). The collection of expressions $e_1, \ldots, e_k$ allows to build collections of values by evaluating $e_1, \ldots, e_k$. Also, we assume the extension of the language with other basic values, and with basic language constructs encoded in a standard way. For instance, we use $e_1; e_2$ to denote the sequential composition of expressions. We use the notation $fn()$ to refer to the set of free names of expressions, defined as expected. Logical conditions, for example database access permissions, are expressed in λ_{DB} with propositions C. Logical propositions are a predicate on a sequence of terms $p(\overline{V})$, a term equality test $V_1 = V_2$, a conjunction $C_1 \wedge C_2$, or an implication $C_1 \Rightarrow C_2$. Note that we separately define values v (which appear in programs) from terms V (which appear in propositions). Terms add to values the field selection construct, in such way allowing propositions to talk about properties of record fields, but the intuition is that terms denote values.

Database constructs are SQL-like. Expression $\mathsf{create}\ t\ :\ \beta_{\overline{\rho}}\ \mathsf{in}\ e$ creates a new database table and binds it to variable t in scope e. The create expression uses the *table type* $\beta_{\overline{\rho}}$, which specifies access control policies for t. A full account of types is given in Section 4; for now it suffices to say that a type $\beta_{\overline{\rho}}$, associates t with a (record) type β for its rows and a set of *permissions* $\overline{\rho}$. As discussed earlier, permissions define access control policies at the level of database entities, based on logical conditions. There are two kinds of permissions: a permission $\mathtt{rd}(m, R)$ specifies that field m can be read only if condition R is deducible from the current knowledge. Similarly, permission $\mathtt{wr}(m, W)$ specifies that field m can be modified only if condition W is deducible from the current knowledge. In a permission $\mathtt{rd}(m, R)$ or $\mathtt{wr}(m, W)$, the (current) database row to which they apply to is denoted by the reserved variable $this$, and any other free variables are considered to be existentially quantified with scope R or W, respectively. For example, $\mathtt{rd}(\mathtt{address}, auth(x) \wedge x = this.id)$ specifies that field address can only be read if there is a value s for which $auth(s)$ holds and $this.id = s$.

Expression $\mathsf{from}\ x\ \mathsf{in}\ t\ \mathsf{where}\ e_1\ \mathsf{select}\ e_2$ specifies a read access to the table t: it returns the collection of all values obtained by applying the select expression e_2 to all

$$
\begin{array}{llll}
e^r ::= e & \text{(Expression)} & \mid \ \mathsf{from}^r_t\ x\ \mathsf{in}\ e\ \mathsf{where}\ e\ \mathsf{select}\ e & \text{(Runtime Select)} \\
\mid \ \mathsf{update}^r_t\ e\ \mathsf{with}\ e & \text{(Runtime Update)} & \mid \ \mathsf{delete}^r_t\ e\ \mathsf{where}\ e & \text{(Runtime Delete)}
\end{array}
$$

Fig. 3. Syntax of λ_{DB} Runtime Expressions.

the rows in t for which the where boolean expression e_1 evaluates to true. Variable x is bound with scope e_1 and e_2.

Expression update x in t where e_1 with e_2 updates the fields in the rows of t that satisfy the where condition e_1 according to the record obtained by evaluating e_2 for every such row. Variable x is bound with scope e_1 and e_2. Expression append e to t adds to t the collection of values obtained by evaluating e. Expression delete x in t where e deletes from t the rows that satisfy the condition e. Variable x is bound with scope e. In examples we specify the database permissions together with entity *invariants* (see, e.g., Fig. 1). Such invariant specifications are syntactic sugar for table type refinements, as the following example illustrates. Consider entity `Friend` in Fig. 1. The corresponding λ_{DB} declaration, letting $\overline{\rho} = \mathtt{rd}(user, Auth(user)), \mathtt{wr}(friend, Auth(user))$, is:
create $Friend\colon \{x\colon [user\colon string;\ friend\colon string] \mid Friends(x.user, x.friend)\}_{\overline{\rho}}\ \ldots$

As other languages with refinement types, λ_{DB} expressions include statements for adding and checking assertions at runtime: assume C specifies that proposition C should be assumed true from the current state on, while assert C checks whether proposition C is true in the current state. For our model to make sense, the use of assume C commands is forbidden to regular users and only allowed in trusted code, accessible to user code through trusted APIs (following the approach of [6]).

3 Operational Semantics

We now present the operational semantics of λ_{DB} and introduce a notion of *access control compliance* for λ_{DB} programs. The semantics is defined using a reduction relation and evaluation contexts [19]. Reduction is defined between *configurations* of the form $(S; C; e)$, where S is a *state*, e is an *expression*, and C is a proposition defining the current *knowledge*. A *reduction step* of the form $(S; C; e) \rightarrow (S'; C'; e')$ means that expression e in state S with knowledge C evolves in one computation step to expression e' in state S' with knowledge C'. State S is a mapping from table names (variables) to collections of basic values, each one annotated with a set of permissions $\overline{\rho}$: $S \triangleq \{t_1 \mapsto \langle v_1 \rangle_{\overline{\rho}_1}, \ldots, t_k \mapsto \langle v_k \rangle_{\overline{\rho}_k}\}$. We use $S(t)$ to refer to the element $\langle v \rangle_{\overline{\rho}}$ such that $t \mapsto \langle v \rangle_{\overline{\rho}} \in S$. We note $dom(S)$ the set of table names defined in state S. *Runtime expressions* (e^r), representing intermediate states in the computation of database operations, are given in Fig. 3. In the following, we use e to denote e^r, where appropriate. *Evaluation contexts* specify the structure of language expressions whose inner expressions are active and may reduce. We write $\mathcal{C}[e]$ to represent the expression obtained by replacing the hole $\cdot$ by e in the evaluation context $\mathcal{C}[\cdot]$. The syntax of evaluation contexts is given in Fig. 4. Reduction relies on an auxiliary notion of knowledge entailment:

Definition 3.1 (Entailment). *Let C and C' be logical propositions. We define $C \vdash C'$ (C entails C') if the proposition $C \Rightarrow C'$ is derivable in classical propositional logic extended with equality over terms in V and the axiom scheme $[\overline{m = v}].m_i = v_i$.*

$$
\begin{array}{lll}
\mathcal{C}[\cdot] ::= \cdot & \text{(Hole)} & \mid \overline{v}, \mathcal{C}[\cdot], \overline{e} \quad \text{(Collection)} \\
\quad \mid \ \mathsf{from}_t^r\ x\ \mathsf{in}\ v\ \mathsf{where}\ \mathcal{C}[\cdot]\ \mathsf{select}\ e & \text{(From)} & \mid \mathsf{let}\ x = \mathcal{C}[\cdot]\ \mathsf{in}\ e \quad \text{(Let)} \\
\quad \mid \ \mathsf{update}_t^r\ v\ \mathsf{with}\ \overline{v\,?\,e:\ e}, \mathcal{C}[\cdot]\,?\,e:\ e, \overline{e} & \text{(Update-If)} & \mid \mathcal{C}[\cdot]\ (v) \quad \text{(Application)} \\
\quad \mid \ \mathsf{update}_t^r\ v\ \mathsf{with}\ \overline{v}, \mathcal{C}[\cdot], \overline{e} & \text{(Update)} & \mid \mathcal{C}[\cdot]\,?\,e:\ e \quad \text{(If)} \\
\quad \mid \ \mathsf{append}\ \mathcal{C}[\cdot]\ \mathsf{to}\ t & \text{(Append)} & \mid \mathcal{C}[\cdot]\ op\ e \quad \text{(Op Left)} \\
\quad \mid \ \mathsf{delete}_t^r\ v\ \mathsf{where}\ \mathcal{C}[\cdot] & \text{(Delete)} & \mid v\ op\ \mathcal{C}[\cdot] \quad \text{(Op Right)} \\
\quad \mid \ [\overline{n = v}, o = \mathcal{C}[\cdot], \overline{m = e}] & \text{(Record)} & \mid \mathcal{C}[\cdot].n \quad \text{(Field)}
\end{array}
$$

Fig. 4. Syntax of Evaluation Contexts.

$$(S; C; \mathsf{true}?e_1{:}e_2) \rightarrow (S; C; e_1)\ \textit{(r-if-true)} \quad (S; C; \mathsf{false}?e_1{:}e_2) \rightarrow (S; C; e_2)\ \textit{(r-if-false)}$$

$$(S; C; \mathsf{let}\ x{=}v\ \mathsf{in}\ e) \rightarrow (S; C; e\{^v/_x\})\textit{(r-let)} \quad (S; C; (\lambda x{:}\tau.e)(v)) \rightarrow (S; C; e\{^v/_x\})\textit{(r-app)}$$

$$\frac{v \neq \star(v')}{(S;\ C;\ [\ldots, n = v, \ldots].n) \rightarrow (S;\ C;\ v)}\textit{(r-field)} \quad \frac{(S;\ C;\ e_1) \rightarrow (S';\ C';\ e_1')}{(S;\ C;\ \mathcal{C}[e_1]) \rightarrow (S';\ C';\ \mathcal{C}[e_1'])}\textit{(r-cont)}$$

$$(S;\ C;\ \mathsf{assume}\ C') \rightarrow (S;\ C \wedge C';\ ())\textit{(r-assume)} \quad \frac{C \vdash C'}{(S;\ C;\ \mathsf{assert}\ C') \rightarrow (S;\ C;\ ())}\textit{(r-assert)}$$

Fig. 5. Reduction Rules for Basic Operations.

For example, we have $(x.m = \mathsf{true} \Rightarrow q(x.m)) \wedge x = [m = \mathsf{true}] \vdash q(\mathsf{true})$.

We may now precisely define the reduction relation on configurations.

Definition 3.2 (Reduction). *Reduction, noted* $(S;\ C;\ e) \rightarrow (S';\ C';\ e')$*, is inductively defined by the rules in Fig. 5, 6, and 7.*

In order to express the notion of compliance with access control policies, we instrument our semantics so that access to values in the state is guarded by the permissions associated to the corresponding tables. We use the notion of classified value to mark the data for which permissions are not entailed by the current knowledge. The rules in Fig. 5 capture the reductions for the conditional expression (*r-if-true*) and (*r-if-false*), let (*r-let*), and application (*r-app*) in a standard way. Rule (*r-field*) states that a record value indexed by a field name reduces to the corresponding field value, provided it is not a classified value. Rule (*r-cont*) allows for reduction to take place internally to a given evaluation context. Rule (*r-assume*) applies to expressions of the form $\mathsf{assume}\ C'$ which reduce to the unit value and add proposition C' to the knowledge in the resulting configuration. By rule (*r-assert*), an expression of the form $\mathsf{assert}\ C'$ reduces, to the unit value, provided that proposition C' is entailed by the current knowledge.

Fig. 6 and 7 present the reduction rules for the operations on database tables. Rule (*r-create*) specifies the creation of a new entry in the state, by associating a fresh table name with an empty collection. Rule (*r-from*) specifies the first step of the evaluation of a from expression by reducing to an intermediate expression. Crucially, the resulting runtime expression from^r takes a *filtered* copy of the values associated with table t in the state, according to the $\mathit{filter}()$ operation defined as follows:

$$\frac{t' \notin dom(S) \cup fn(e)}{(S; C;\ \mathsf{create}\ t : \beta_{\overline{\rho}}\ \mathsf{in}\ e) \longrightarrow (S, t' \mapsto \langle\emptyset\rangle_{\overline{\rho}}; C;\ e\{t'/t\})}(\textit{r-create})$$

$$\frac{S(t) = \langle\overline{v}\rangle_{\overline{\rho}} \qquad \overline{v}' = \mathit{filter}(\overline{v})^{\overline{\rho}}_{C}}{(S; C;\ \mathsf{from}\ x\ \mathsf{in}\ t\ \mathsf{where}\ e_1\ \mathsf{select}\ e_2) \longrightarrow (S; C;\ \mathsf{from}^r_t\ x\ \mathsf{in}\ \overline{v}'\ \mathsf{where}\ \overline{e_1\{v'/x\}}\ \mathsf{select}\ e_2)}(\textit{r-from})$$

$$\frac{S(t) = \langle\overline{v}\rangle_{\overline{\rho}} \qquad \overline{v}' = \mathit{filter}(\overline{v})^{\overline{\rho}}_{C}}{(S; C;\ \mathsf{update}\ x\ \mathsf{in}\ t\ \mathsf{where}\ e_1\ \mathsf{with}\ e_2) \longrightarrow (S; C;\ \mathsf{update}^r_t\ \overline{v}\ \mathsf{with}\ \overline{(e_1\ ?\ e_2\ :\ [])\{v'/x\}})}(\textit{r-update})$$

$$\frac{S(t) = \langle\overline{v}\rangle_{\overline{\rho}} \qquad \overline{v}' = \mathit{filter}(\overline{v})^{\overline{\rho}}_{C}}{(S; C;\ \mathsf{delete}\ x\ \mathsf{in}\ t\ \mathsf{where}\ e) \longrightarrow (S; C;\ \mathsf{delete}^r_t\ \overline{v}\ \mathsf{where}\ \overline{e\{v'/x\}})}(\textit{r-delete})$$

$$\frac{\mathit{ok2write}(\overline{v})^{\overline{\rho}}_{C}}{(S, t \mapsto \langle\overline{u}\rangle_{\overline{\rho}}; C;\ \mathsf{append}\ \overline{v}\ \mathsf{to}\ t) \longrightarrow (S, t \mapsto \langle\overline{u}, \overline{v}\rangle_{\overline{\rho}}; C;\ ())}(\textit{r-append})$$

Fig. 6. Reduction Rules for Table Operations.

$$\frac{\overline{u} = \{v'_k \mid v_k = \mathsf{true}\}}{(S; C;\ \mathsf{from}^r_t\ x\ \mathsf{in}\ \overline{v}'\ \mathsf{where}\ \overline{v}\ \mathsf{select}\ e_2) \longrightarrow (S; C;\ \overline{e_2\{u/x\}})}(\textit{r-from}^r)$$

$$\frac{\forall_i\ (e^3_i = e^1_i \wedge u_i = \mathsf{true}) \vee (e^3_i = e^2_i \wedge u_i = \mathsf{false})}{(S; C;\ \mathsf{update}^r_t\ \overline{v}\ \mathsf{with}\ \overline{u\ ?\ e^1\ :\ e^2}) \longrightarrow (S; C;\ \mathsf{update}^r_t\ \overline{v}\ \mathsf{with}\ \overline{e^3})}(\textit{r-update-if}^r)$$

$$\frac{\overline{u} = \overline{v} \bullet \overline{v}' \qquad \mathit{ok2update}(\overline{v}, \overline{v}')^{\overline{\rho}}_{C}}{(S, t \mapsto \langle u'\rangle_{\overline{\rho}}; C;\ \mathsf{update}^r_t\ \overline{v}\ \mathsf{with}\ \overline{v}') \longrightarrow (S, t \mapsto \langle\overline{u}\rangle_{\overline{\rho}}; C;\ ())}(\textit{r-update}^r)$$

$$\frac{\overline{u} = \{v_k \mid v'_k = \mathsf{false}\} \qquad \mathit{ok2write}(\{v_k \mid v'_k = \mathsf{true}\})^{\overline{\rho}}_{C}}{(S, t \mapsto \langle u'\rangle_{\overline{\rho}}; C;\ \mathsf{delete}^r_t\ \overline{v}\ \mathsf{where}\ \overline{v}') \longrightarrow (S, t \mapsto \langle\overline{u}\rangle_{\overline{\rho}}; C;\ ())}(\textit{r-delete}^r)$$

Fig. 7. Reduction Rules for Runtime Expressions.

Definition 3.3 (Filtering). *Given a set of permissions $\overline{\rho}$, a proposition C, and a record $r = [\overline{m = v}]$, we define filtering of r under $C, \overline{\rho}$, by $\mathit{filter}(r)^{\overline{\rho}}_{C} \triangleq [\overline{m = v'}]$ where:*

$$v'_i = \begin{cases} v_i & \textit{if exists } \mathtt{rd}(m_i, R) \in \overline{\rho} \textit{ and } \theta \textit{ such that } C \vdash \theta(R\{r/this\}) \\ \star(v_i) & \textit{otherwise} \end{cases}$$

We set $\mathit{filter}(v_1, \ldots, v_n)^{\overline{\rho}}_{C} = \mathit{filter}(v_1)^{\overline{\rho}}_{C}, \ldots, \mathit{filter}(v_n)^{\overline{\rho}}_{C}$.

The filtering operation marks a value v_i in a record field as *classified* if no instance of its read permissions is derivable from the current knowledge C, replacing v_i by $\star(v_i)$ in the resulting record. A substitution θ (a finite function from variables to terms) is used to instantiate all free variables in a permission condition by closed values (except for the reserved variable $this$). From now on, we use $\theta(C)$ assuming that the domain of θ is the set of free variables in C, except $this$. Filtering causes a program to get stuck if it attempts to select a classified value from a record read from a table later on in the computation.

In the runtime counterpart of expression from x in t where e_1 select e_2, expression e_1 is expanded to a collection of expressions, where each element ($e_1\{v'_i/x\}$) instantiates the cursor variable x with one of the filtered rows (v'_i) of table t. Notice that the from^r expression freezes the current (filtered) state of table t, so as to use it when producing the final result. Rule (*r-from*^{r}) applies to a from^r expression where all conditional expressions are values, and reduces to a collection of expressions, obtained by replacing the cursor variable x by each one of the selected rows in the select expression e_2.

By rule (*r-update*), an update expression reduces to a runtime expression update^r, that expresses the modifications to the selected rows of t, via a collection of conditional constructs ($e_1?e_2{:}[]$), where the cursor variable x is replaced by the filtered values of table t. If the condition e_1 yields true, the modified field values are computed by expression e_2, otherwise the result is an empty record denoting that no modification is to be performed in that particular row. Rule (*r-update-if*^{r}), is applied after the evaluation of all conditions, and performs the corresponding selections. This three-step evaluation ensures the expected semantics where conditions are all evaluated first. Finally, rule (*r-update*^{r}) actually updates the table in the resulting state. The record update operation below is used to update the collection associated to table name t in the state. It takes two records r,r' and produces a record based on the first argument, replacing its field values with the values of the second record whenever they exist.

Definition 3.4 (Record Update). *Let $r = [\overline{m = v}]$ and $r' = [\overline{m' = v'}]$ be two records with $\overline{m'} \subseteq \overline{m}$. The update of record r by r', noted $r \bullet r'$, is defined by*

$$r \bullet r' \triangleq [\overline{m = u}] \text{ where } u_i = (\text{if } m_i \in \overline{m}' \text{ then } r'_{m_i} \text{ else } r_{m_i})$$

We set $\overline{r} \bullet \overline{r'} = r_1 \bullet r'_1, \ldots, r_n \bullet r'_n$.

For example $[pwd = foo, uid = 9] \bullet [uid = 0] = [pwd = foo, uid = 0]$.

Rule (*r-delete*) specifies that a delete expression reduces to the runtime expression delete^r, in which the where expression is expanded into a collection of boolean tests, again instantiating the cursor variable x with the filtered records $\overline{v'}$. Rule (*r-delete*^{r}) updates the values in table t in the resulting state, by keeping only the ones whose corresponding test yields false. Rule (*r-append*) reduces to a configuration where the collection associated with table name t is imperatively augmented with values $\overline{v}$. The operations update, delete, and append depend on the runtime verification that the current knowledge entails the necessary write permissions. Rule update^r expression depends on test $ok2update()$ that checks only the modified fields in a record when compared with the original row, while rules (*r-delete*^{r}) and (*r-append*^{r}) depend on the test $ok2write()$ which checks permissions for all fields in the table rows.

Definition 3.5 (Write and Update Permission Checks). *Given $r = [\overline{m = u}]$ and $r' = [\overline{m' = v}]$ (with $\overline{m'} \subseteq \overline{m}$), a set of permissions $\overline{\rho}$ for r, and a proposition C, we define the write and update permission checks, $ok2write(r)^{\overline{\rho}}_C$, and $ok2update(r, r')^{\overline{\rho}}_C$ by:*

$$ok2write(r)^{\overline{\rho}}_C \triangleq \forall_{m_i \in \overline{m}}\ \exists W_i, \theta_i(\mathtt{wr}(m_i, W_i) \in \overline{\rho} \text{ and } C \vdash \theta_i(W_i\{r/this\}))$$

$$ok2update(r, r')^{\overline{\rho}}_C \triangleq$$
$$\forall_{m_i \in \overline{m'}}\ (r_{m_i} = r'_{m_i}) \vee \exists W_i, \theta_i(\mathtt{wr}(m_i, W_i) \in \overline{\rho} \text{ and } C \vdash \theta_i(W_i\{r'/this\}))$$

We set $ok2write(\overline{v})^{\overline{\rho}}_C = ok2write(v_1)^{\overline{\rho}}_C \wedge \ldots \wedge ok2write(v_n)^{\overline{\rho}}_C$,
and $ok2update(\overline{v}, \overline{u})^{\overline{\rho}}_C = ok2update(v_1, u_1)^{\overline{\rho}}_C \wedge \ldots \wedge ok2update(v_n, u_n)^{\overline{\rho}}_C$.

$$
\begin{array}{lll}
\beta, \phi ::= \texttt{unit} & \text{(Unit Type)} & \\
\quad | \quad \texttt{bool} & \text{(Boolean Type)} & \\
\quad | \quad \overline{[m : \beta]} & \text{(Record Type)} & \\
\quad | \quad \{x : \beta \mid C\} & \text{(Refinement Type)} &
\end{array}
\qquad
\begin{array}{lll}
\tau, \sigma ::= \beta & \text{(Basic Type)} & \\
\quad | \quad \beta^* & \text{(Collection Type)} & \\
\quad | \quad \Pi x : \tau.\, \tau & \text{(Dependent Function Type)} &
\end{array}
$$

Fig. 8. Syntax for Types.

We may now define the notion of error for λ_{DB} configurations.

Definition 3.6 (Error). *A configuration* $(S; C; e)$ *is an* error *if* e *is not a value and there are no* S', C', e' *such that* $(S; C; e) \rightarrow (S'; C'; e')$.

Notice that a configuration $(S; C; e)$ immediately attempting to select a field of a record containing a hidden (classified) value is an error, since by the premise of (*r-field*), it has no reduction. Given that classified values are only introduced in data access primitives by filtering out data in fields for which no read permission is available, we define:

Definition 3.7 (Data Access Control Compliance). *A configuration is* data access control compliant *if no computation from it gets stuck in a field selection* (*r-field*)*, update, delete, or append operation, due to a* $ok2write()$ *or* $ok2update()$ *test failure.*

It is then clear that programs that do not get into errors are in particular access control compliant. In the next section, we introduce a type system that statically ensures that well-typed programs do not get into errors, and are therefore access control compliant.

4 Type System

In this section we present our type system, which ensures that well-typed programs are data access control compliant. The syntax of types is defined in Fig. 8. We have basic types (β): `unit`, `bool`, the record type $[m_1 : \beta_1, \ldots, m_k : \beta_k]$, and refinement types $\{x : \beta \mid C\}$, which capture values of type β for which the proposition C holds (x is bound with scope C). Types also include collection types β^* which type collections of values of type β, and dependent function types $\Pi x : \tau.\, \sigma$, the type of functions that given a value x of type τ return a value of type σ, where x may occur on σ. As usual, the standard function type $\tau \rightarrow \sigma$ is represented by $\Pi x : \tau.\, \sigma$, where x does not occur in σ. Notice that we forbid collections of functions, collections of collections, etc, for simplicity. We also introduce table types, denoted $\beta_{\overline{\rho}}$ to classify table names. Recall that table names are imperatively bound to collection of values of type β, and have their contents are guarded by a set of permissions $\overline{\rho}$.

We now present our typing relation. A typing judgment of the form $\Delta \vdash e : \tau$ says that expression e has type τ under environment Δ. Also, we use $\Delta \vdash C$ to say that knowledge C is logically entailed from the knowledge in environment Δ (see below). We introduce an auxiliary type constructor, not expected to appear on source programs, but needed to type records where some fields may contain secured data. Such types, called projected types, have the form $\beta\lfloor_{\overline{m}}$, where β is a record type or an (hereditary) refinement of a record type, and are only used to type the query "cursor" in the scope of database table operations. Intuitively, $\beta\lfloor_{\overline{m}}$ means the same as β, but selection on a

"classified" field (a field not in $\overline{m}$) is not allowed in well-typed programs. Crucially, types containing projected types as subexpressions are not allowed, so that the projection construction in only allowed to occur at the top level of any type. This condition is important to block illegal information flows out of trusted where and select clauses. In the sequel, we range both types τ and projected types $\beta\lfloor_{\overline{m}}$ using μ.

Type declarations, ranged over by γ, are assignments of types to variables, defined as either $x : \tau$ (normal type) or as $x : \tau\lfloor_{\overline{m}}$ (projected type) or as $t : \beta_{\overline{\rho}}$ (table type). For example, $x : [\texttt{name} : \beta_1, \texttt{email} : \beta_2, \texttt{address} : \beta_3]\lfloor_{\texttt{name,email}}$ is a type declaration that specifies that the only fields accessible in variable x are `name` and `email`, while field `address` is not accessible. A typing environment, ranged over by Δ, is a sequence of typing declarations $\gamma_1, \ldots, \gamma_k$. A well-formed typing environment satisfies a domain closure property on type declarations (from left to right). We say typing environment Δ is well-formed if for all γ such that $\Delta = \Delta', \gamma, \Delta''$ and $\gamma = x : \tau$ or $\gamma = x : \tau\lfloor_{\overline{m}}$ or $\gamma = x : \beta_{\overline{\rho}}$ then $x \notin fn(\Delta')$, where there is a notion of *free names* of environments, declarations, and types (we use $fn()$, taking into account names in the domain and in the types). From this point on, we assume typing environments are always well-formed. Also, we use Δ_π to denote the environment obtained by deleting from Δ all identifiers which are assigned non-basic types. To define the knowledge of a type environment, we introduce the auxiliary notion of term environment. This is the same as the notion of environment, but where *term declarations* may assign types to *terms* V (see Fig. 2), not just to variables. So a type environment is also a term environment. Given a term environment Δ, we may consider the knowledge it expresses about the terms it specifies, as a set (taken as the conjunction) of propositions.

Definition 4.1 (Knowledge). *The knowledge of a term declaration γ, noted $kn(\gamma)$, is inductively defined on types by:*

$$kn(V{:}\{x{:}\beta \mid C\}) \triangleq \{C\{^V/_x\}\} \cup kn(V{:}\beta) \qquad kn(V{:}[\overline{m{:}\beta}]) \triangleq \bigcup_{m_i \in \overline{m}} kn(V.m_i{:}\beta_i)$$
$$kn(V{:}\{x{:}\beta \mid C\}\lfloor_{\overline{m}}) \triangleq \{C\{^V/_x\}\} \cup kn(V{:}\beta\lfloor_{\overline{m}}) \qquad kn(V{:}[\overline{m{:}\beta}]\lfloor_{\overline{n}}) \triangleq \bigcup_{m_i \in \overline{n}} kn(V.m_i{:}\beta_i)$$

and as $kn(\gamma) \triangleq \emptyset$ for other types. Then $kn(\Delta)$ is given by $kn(\Delta, \gamma) = kn(\Delta) \cup kn(\gamma)$, and $kn(\emptyset) = \emptyset$. We often identify the set $kn(\Delta)$ with the conjunction of its elements.

Definition 4.2 (Derivable Knowledge). *Given a term environment Δ, formula C is derivable knowledge from Δ, noted by $\Delta \vdash C$, if $kn(\Delta) \vdash C$.*

Logical entailment has been defined in Definition 3.1. We can verify that knowledge is preserved by term substitution, that is, $kn(\Delta)\{^V/_x\} = kn(\Delta\{^V/_x\})$, and we also have that $\Delta \vdash V{:}\mu$ implies $\Delta \vdash kn(V{:}\mu)$. We can now define:

Definition 4.3 (Typing). *Typing is expressed by judgment $\Delta \vdash e : \mu$, stating expression e is well-typed by μ in environment Δ. Typing rules are given in Figs. 9, 10, and 11.*

We first discuss the typing rules that do not concern database operations, depicted in Fig. 9. Rules (*t-assert*), (*t-assume*), (*t-refine*), (*t-term-refine*), and (*t-unrefine*) express standard principles in refinement type theories (see, e.g., [15]), with some simplifications, due to the absence of subtyping in our presentation. Rule (*t-assert*) checks if the environment knowledge supports the specified proposition. Rule (*t-assume*) (remember

$$\frac{\Delta \vdash C}{\Delta \vdash \mathsf{assert}\ C : \mathtt{unit}}(\textit{t-assert}) \qquad \Delta \vdash \mathsf{assume}\ C : \{\_ : \mathtt{unit} \mid C\}\ (\textit{t-assume})$$

$$\Delta \vdash () : \mathtt{unit}\ (\textit{t-unit}) \qquad \frac{\Delta \vdash V\ :\ \beta \qquad \Delta \vdash C(V)}{\Delta \vdash V\ :\ \{x : \beta \mid C(x)\}}(\textit{t-term-refine})$$

$$\frac{\Delta \vdash e\ :\ \beta \qquad \Delta, x : \beta \vdash C(x)}{\Delta \vdash e\ :\ \{x : \beta \mid C(x)\}}(\textit{t-refine}) \qquad \frac{\Delta \vdash e\ :\ \{x : \beta \mid C(x)\}}{\Delta \vdash e\ :\ \beta}(\textit{t-unrefine})$$

$$\frac{\Delta \vdash e : \Pi x : \tau.\ \sigma \qquad \Delta \vdash v : \tau}{\Delta \vdash e\,(v) : \sigma\{v/x\}}(\textit{t-app}) \qquad \frac{\Delta \vdash e_1 : \sigma \qquad \Delta, x : \sigma \vdash e_2 : \tau}{\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}(\textit{t-let})$$

$$\frac{\Delta, x : \tau \vdash e : \sigma}{\Delta \vdash \lambda x : \tau.e : (\Pi x : \tau.\ \sigma)}(\textit{t-fun}) \qquad \frac{op : \tau_1 \to \tau_2 \to \sigma \qquad \forall_{i \in 1,2}\ \Delta \vdash e_i : \tau_i}{\Delta \vdash e_1\ op\ e_2 : \sigma}(\textit{t-op})$$

$$\frac{\begin{array}{c}\Delta \vdash e_1 : \{b : \mathtt{bool} \mid C(b)\} \\ \Delta, \_ : \{\_ : \mathtt{unit} \mid C(\mathsf{true})\} \vdash e_2 : \tau \qquad \Delta, \_ : \{\_ : \mathtt{unit} \mid C(\mathsf{false})\} \vdash e_3 : \tau\end{array}}{\Delta \vdash e_1\,?\,e_2 :\ e_3 : \tau}(\textit{t-if})$$

$$\Delta, x : \tau, \Delta' \vdash x : \tau\ (\textit{t-id}) \qquad \frac{\forall_i\ \Delta \vdash e_i : \beta_i}{\Delta \vdash [\overline{m = e}] : [\overline{m : \beta}]}(\textit{t-record}) \qquad \frac{\forall_i\ \Delta \vdash e_i : \beta}{\Delta \vdash \overline{e} : \beta^*}(\textit{t-collection})$$

$$\frac{\Delta \vdash e : [\ldots, n : \beta, \ldots]}{\Delta \vdash e.n : \beta}\ (\textit{t-field}) \qquad \frac{\Delta \vdash v : [\ldots, n : \beta, \ldots]\lfloor_{\overline{m}} \qquad n \in \overline{m}}{\Delta \vdash v.n : \beta}\ (\textit{t-fieldProj})$$

$$\frac{\forall_i\ ((v_i = \star(u_i) \wedge m_i \notin \overline{n}) \vee \Delta \vdash v_i : \beta_i)}{\Delta \vdash [\overline{m = v}] : [\overline{m : \beta}]\lfloor_{\overline{n}}}(\textit{t-recProj}) \qquad \Delta, x : \tau\lfloor_{\overline{m}}, \Delta' \vdash x : \tau\lfloor_{\overline{m}}\ (\textit{t-idProj})$$

$$\frac{\Delta \vdash x : \tau\lfloor_{\overline{m}} \qquad \mathit{fields}(\tau) = \overline{m}}{\Delta \vdash x : \tau}\ (\textit{t-allFields})$$

Fig. 9. Typing Rules (I).

that assume is only to be used in trusted code, not user code) types the assume witness (with unit type) with its logical refinement, which may then be added to the current knowledge (namely, via a let $x = \mathsf{assume}\ C\ \mathsf{in}\ \ldots$).

We now consider the basic typing rules for database operations (see Fig. 10). Typing rules for related runtime expressions follow similar lines and are shown in Fig. 11. Rule (*t-from*) specifies that a from expression is well-typed if the environment knowledge entails the permissions needed to access the data in the table (I, J, K, L are sets of record field labels). The where expression e_1 returns a boolean b such that $C(x, b)$ for the given record x. Thus if x is selected by the test e_1, we know $C(x, \mathsf{true})$ holds. Notice that e_1 itself is only allowed to use table fields m_j ($m_j \in J$) for which the appropriate read permissions R_{m_j} are entailed by the current knowledge Δ. Using this additional piece of knowledge ($C(x, \mathsf{true})$), taking into account that the result of the where test is true for the selected rows, the set of permissions R_{m_k}, for fields $m_k \in K$, is derived. Notice that the refinement predicate obtained for the where expression (i.e., $C(x, b)$) carries information about the actual data being selected. This allows us to capture the intended row-level access control conditions, as all necessary read permissions are valid for each

$$\frac{\begin{array}{c}\Delta(t) = \beta_{\{\mathtt{rd}(m_i, R_{m_i}) \mid m_i \in I\} \cup \overline{\rho}} \qquad J, K \subseteq I \\ \Delta \vdash \bigwedge_{m_j \in J} \theta_j(R_{m_j}) \qquad \Delta_\pi, x : \beta\lfloor_J \vdash e_1 : \{b : \mathtt{bool} \mid C(x, b)\} \\ \Delta \vdash C(\mathit{this}, \mathsf{true}) \implies \bigwedge_{m_k \in K} \theta_k(R_{m_k}) \\ \Delta, x : \beta\lfloor_K, \_ : \{\_ : \mathtt{unit} \mid C(x, \mathsf{true})\} \vdash e_2 : \phi\end{array}}{\Delta \vdash \mathsf{from}\ x\ \mathsf{in}\ t\ \mathsf{where}\ e_1\ \mathsf{select}\ e_2 : \phi^*}(\textit{t-from})$$

$$\frac{\begin{array}{c}\Delta(t) = \beta_{\{\mathtt{rd}(m_i, R_{m_i}) \mid m_i \in I\} \cup \{\mathtt{wr}(m_l, W_{m_l}) \mid m_l \in L\} \cup \overline{\rho}} \qquad J, K \subseteq I \qquad L \subseteq K \\ \Delta \vdash \bigwedge_{m_j \in J} \theta_j(R_{m_j}) \qquad \Delta_\pi, x : \beta\lfloor_J \vdash e_1 : \{b : \mathtt{bool} \mid C(x, b)\} \\ \Delta \vdash C(\mathit{this}, \mathsf{true}) \implies \bigwedge_{k \in K} \theta_{m_k}(R_{m_k}) \\ \beta = \{r : \tau \mid I(r)\} \quad I(r) \vdash H(r) \quad H(r) \wedge U(r) \vdash I(r) \quad \phi = \{r : |\tau|_K \mid U(r)\} \\ D(y) = (\bigwedge_{m_l \in K - L} y.m_l = x.m_l) \wedge \bigwedge_{m_l \in L} \theta_{m_l}(W_{m_l}\{y/\mathit{this}\}) \\ \Delta, x : \beta\lfloor_K, \_ : \{\_ : \mathtt{unit} \mid C(x, \mathsf{true})\} \vdash e_2 : \{y : \phi \mid D(y)\}\end{array}}{\Delta \vdash \mathsf{update}\ x\ \mathsf{in}\ t\ \mathsf{where}\ e_1\ \mathsf{with}\ e_2 : \mathtt{unit}}(\textit{t-update})$$

$$\frac{\begin{array}{c}\Delta(t) = \beta_{\{\mathtt{rd}(m_j, R_{m_j}) \mid m_j \in J\} \cup \{\mathtt{wr}(m_l, W_{m_l}) \mid m_l \in L\} \cup \overline{\rho}} \qquad \mathit{fields}(\beta) = L \\ \Delta \vdash \bigwedge_{m_j \in J} \theta_j(R_{m_j}) \qquad \Delta_\pi, x : \beta\lfloor_J \vdash e : \{b : \mathtt{bool} \mid C(x, b)\} \\ \Delta \vdash C(\mathit{this}, \mathsf{true}) \implies \bigwedge_{m_l \in L} \theta_l(W_{m_l})\end{array}}{\Delta \vdash \mathsf{delete}\ x\ \mathsf{in}\ t\ \mathsf{where}\ e : \mathtt{unit}}(\textit{t-delete})$$

$$\frac{\begin{array}{c}\Delta(t) = \beta_{\{\mathtt{wr}(m_l, W_{m_l}) \mid m_l \in L\} \cup \overline{\rho}} \qquad \mathit{fields}(\beta) = L \\ \Delta \vdash e : \{x : \beta \mid \bigwedge_{m_l \in L} \theta_l(W_{m_l}\{x/\mathit{this}\})\}^*\end{array}}{\Delta \vdash \mathsf{append}\ e\ \mathsf{to}\ t : \mathtt{unit}}(\textit{t-append})$$

$$\frac{\Delta, t : \beta_{\overline{\rho}} \vdash e : \tau \qquad \beta = \{r : \beta_r \mid I(r)\} \quad \beta_r \text{ is a record type}}{\Delta \vdash \mathsf{create}\ t : \beta_{\overline{\rho}}\ \mathsf{in}\ e : \tau}(\textit{t-create})$$

Fig. 10. Typing Rules (II).

selected row. For soundness, we type the test in the pure part of the environment Δ_π. This ensures that computation of where clauses will never generate new knowledge, even if they may generate side effects.

Notice that the type of the cursor x is projected to the set of accessible fields when typing the where and select expressions, so that only the fields for which permissions are entailed may be selected (see Rule (*t-fieldProj*)). Also, to derive the permissions, we type the cursor identifier x with the (row) element type for the table (β), projected to the accessible fields (either m_j or m_k). The typing of the select expression ensures that ϕ is the type of the values returned; in general, ϕ may implicitly include information on the invariants of entity t as a refinement.

Rule (*t-fieldProj*) types the field access to an identifier which type declaration is projected in a set of field names and where this set contains the accessed field. In rule (*t-allFields*), an environment that specifies a projected type declaration for identifier n types it with the (unrestricted) type τ, provided the projection is over all fields of type τ — we use $\mathit{fields}(\tau)$ to denote the set of fields of the record type τ, possibly occurring under a refinement type. The combined use of rules (*t-allFields*) and (*t-fieldProj*) ensures that a record value typed by a projected type can only be used as a non protected

record when the projection is over the whole set of fields. Otherwise, the only admissible behavior on such a value will be to select a field in the projection set. This ensures a tight control on the information flow of secured data. In particular, classified (high) values $\star(v)$ will never be leaked outside the context of a database operation, such as a where or a select computation, since protected types are not used elsewhere, due to our typing and syntactic constraints.

Rule (*t-update*) implements a reasoning similar to (*t-from*), as far as the where test is concerned, with fields in J selected for safe reading, based on available read permissions R_{m_j}. However, to type the with clause e_2, we must check: (1) that e_2 copies without modification ($y.m_l = x.m_l$) the fields in $K - L$ for which write permissions are not deduced, and (2) that write permissions W_{m_k} are currently entailed for all potentially modified fields m_k. In general, write permissions W_{m_k} may refer to readable fields (allowed by R_{m_l}), where $L \subseteq K$ (only readable fields may be updated), but not to other (classified) fields. So, we require in the rule that the permissions $W_{m_l}(r)$ may only refer to fields of r in K. All these conditions explain the refinement $D(y)$ of ϕ. To ensure that the updated row would satisfy the table type $\beta = \{r : \tau \mid I(r)\}$ (expressing invariant $I(r)$) we verify, through a frame reasoning, that the conjunction of the properties of the update records produced by e_2 together with the properties of the classified part imply the table invariant $I(r)$ for the updated record r. So, we require that $H(r)$ only refers to fields of τ not in K and $U(r)$ only refers to fields of τ in K. Notice that the base record type of ϕ is the same of β, but with fields out of K removed (noted $|\tau|_K$). All these conditions ensure that the update $v_3 = v_1 \bullet v_2$ in rule (*r-update-if*) is well defined, and that neither invariants nor write permissions are violated at runtime.

The remaining rules follow similar ideas: (*t-delete*) verifies that write permissions are available for all fields of all records selected by the where test, and (*t-append*) requires write access to all table fields. We now introduce well-typed configurations.

Definition 4.4 (Well-typed Configuration and State). *A configuration $(S; C; e)$ is well-typed in environment Δ (noted $\Delta \vdash (S; C; e)$ if (a) $\Delta \vdash e : \tau$; (b) $\Delta \vdash S$; and (c) for all $C_i \in kn(\Delta)$, $C \vdash C_i$. We also define $\Delta \vdash S$ by for all $(t \mapsto \langle \overline{v} \rangle_{\overline{\rho}}) \in S$ we have $\Delta(t) = \beta_{\overline{\rho}}$ and for all $v_i \in \overline{v}$, $\Delta \vdash v_i : \beta$.*

Notice that (c) states that the runtime condition C (the log) is stronger that the static knowledge entailed by Δ. This fact is used in our proofs to express that any statically verified condition (assertions, permissions) also holds at runtime.

We now present our main results, which ensure that programs that go through our typing rules are access control compliant (in addition to being of course error (stuck) free in the usual sense). The main statements are Theorem 4.6 (Type Preservation) — well-typing is invariant under reduction — Theorem 4.7 (Progress) — well-typed expressions are either values or have a reduction — and Corollary 4.8 — well-typed expressions comply with access control policies. Detailed proofs may be found in [9]. We first present our type preservation result, which relies on two substitution lemmas.

Lemma 4.5 (Substitution).

1. *(Entailment) Let $\Delta, x : \mu, \Delta' \vdash C$. If $\Delta \vdash V : \mu$ then $\Delta, (\Delta'\{^V/_x\}) \vdash C\{^V/_x\}$.*
2. *(Typing) Let e be an expression where $\Delta, x : \mu, \Delta' \vdash e : \sigma$. If $\Delta \vdash v : \mu$ then $\Delta, (\Delta'\{^v/_x\}) \vdash e\{^v/_x\} : \sigma\{^v/_x\}$.*

Theorem 4.6 (Type Preservation). *Let* $\Delta \vdash (S; C; e)$. *If* $(S; C; e) \rightarrow (S'; C'; e')$ *then there is* Δ' *such that* $\Delta, \Delta' \vdash (S', C', e')$.

Theorem 4.6 says that any reduction step in well-typed configuration leads to a well-typed configuration, where the typing of the final configuration is possibly extended so as to capture added knowledge (e.g., in case of an assume) or new locations of tables (via create) in the state. Our second result states that a closed well-typed configuration $(S; C; e)$ either has a value as its distinguished expression e, or has a reduction (it is not stuck). In particular, it is not an error.

Theorem 4.7 (Progress). *Let* $\Delta \vdash (S; C; e)$ *and* $fn(e) \subseteq dom(S)$. *Then either* e *is a value or* $(S; C; e) \rightarrow (S'; C'; e')$.

Theorem 4.7 states closed well-typed programs never get stuck on a expression which is not a value, and thus, in particular, well-typed configurations never get stuck on an assert C statement. Also notice that access to a classified record value is never attempted performed by well-typed programs, and safety of database updates, deletes, and appends, is also ensured by the validity of runtime control assertions such as $ok2write()$, which ultimately depend on assertion checking. By Theorems 4.6 and 4.7 we conclude:

Corollary 4.8 (Data Access Control Compliance). *Let* $\Delta \vdash (S; C; e)$ *and* $fn(e) \subseteq dom(S)$. *Then* $(S; C; e)$ *is data access control compliant.*

Corollary 4.8 tells us, in a technically precise way, that well-typed configurations never attempt to read data from tables which is forbidden by the prescribed policies, neither store in tables data that violates the prescribed policies or table invariants.

5 Discussion and Related Work

Refinement types were introduced in [13] in the context of ML type theory. Recently, Gordon and co-authors have developed a general theory of refinement types for a concurrent λ-calculus [15]. Their approach is applied in [6], where refinement types are interpreted as logic assertions which are used to verify authentication/authorization properties in security protocols. Our model builds on the general framework of refinement types, but does not seem naturally expressible within it; in fact, a key ingredient of our approach is the integration of refinements with coarse-grained typing principles for database operations, allowing access policies to selectively depend on the stored data, on a row-level basis, as required in realistic scenarios. We have no perspective on how to model our typed language in other related languages, including Fine [17], mainly because of the need to tightly integrate the constraints needed to statically type-check the trusted from and update constructs, parametric on general filtering where tests.

Several (proposals of) programming and/or modeling languages integrating data operations into programs — often in the form of SQL-like operations — have been put forward recently; examples include LINQ [16], Cω [8], Links [11], Ur/Web [10], Dminor [7], and M; some of these works have tackled security issues. Our work shares the same general goals with [10]. However, the underlying approaches are very different. In [10] access policies are defined as queries and programs are checked (using symbolic

evaluation techniques) so as to guarantee any data exchanged with the database is contained in the result set of some policy, while access control policies may depend on the actual data and on the knowledge held by the user performing the query — captured by *known*, a distinguished predicate. In our approach access control policies also depend on the actual data; they are expressed in terms of arbitrary logical expressions which can capture, for instance, the knowledge held by the user or some data relationship, thus introducing extra flexibility. Another fundamental difference between our work and [10] lies on the conception of access control itself. The approach in [10] enforces a strong distinction between the access control capabilities of the (trusted) server and those of its (untrusted) clients. In our work, thanks to refinement types, access control can be treated in a more uniform way: the actual access control permissions of a participant (server or clients) depends on the knowledge currently available to it. This is reflected in the handling of where clauses: in [10] the data by a where clause is not subject to access control checks, which allows for queries that implicitly leak non-accessible information while in our approach the where test code is subject to access control checks in a uniform way, which excludes queries that access classified information, in the precise sense of Corollary 4.8.

Dminor [7] combines refinement types and expressions enforcing dynamic type tests. Although a basic form of select expressions is expressible in Dminor (by means of an accumulator construct), it does not support other database-like expressions nor offers a simple way of enforcing fine-grained access control permissions over data entities, two of the distinguishing features of λ_{DB}. The approach of [7] is related to the work in RFC in [6], which we reviewed above. Links [11] is a typed functional programming language for web applications. In [5] a secure compilation strategy for Links is formalized by a concurrent λ-calculus, endowed with a refinement type system, but not addressing data management operations. SELinks [12] extends Links with label-based security policies. In SELinks security labels are associated to objects which contain sensible data; a labeled object is not accessible. Policy enforcement functions are used to safely "unlabel" objects, thus implementing the semantics associated to the security labels. A type system [18] ensures the correct mediation between application code and the policy enforcement functions. Our focus is on data access control, based on logical conditions on the data itself, where permissions are directly taken in account while typing database primitives, by means of refinement types, allowing for fine-grained policies to be directly verified in application level SQL-like database manipulation code.

6 Concluding Remarks

We have presented a type-based approach to statically enforce access control to persistent data in data-centric software systems such as web applications. We believe that our proposal provides a useful mechanism to enforce the preservation of protection (in the general sense of [1]) between application-level security and database-level security management. Our technical development is based on a core language λ_{DB}, which includes comprehensive SQL-like operations, and is equipped with a refinement type system. Refinement types are combined with access permissions, allowing the system to control unsecure information flows across database operations. Although simple to

use, at least in principle, our approach is expressive enough to enforce access control policies at a very fine-grained level, in particular allowing security constraints to dynamically depend on the stored data, as is often the case in real applications. Our main results certify that our type system excludes error configurations, namely systems that violate access control policies, in a technically precise sense. In future work, we intend to extend our model with more sophisticated permission and access control concerns. For simplicity and usability, our refinements logic is a simple classical logic, where deduced facts are monotonically accumulated, while database contents keeps changing overtime. Nevertheless, this approach is already very useful and consistent with the scenario of web-based applications, where all the state resides in the database, and operations are implemented by independent short-running requests. Our type system ensures that no data access violations may occur in a transaction, given the currently visible database stored data and related policies. Nevertheless, it would be interesting to study the adoption in our framework of more expressive logics (e.g., [3, 14, 4]), and to research the interplay between refinement types and types for information flow [2].

Acknowledgements. We thank Carnegie Mellon—PT INTERFACES 44-2009-12, CITI, and OutSystems SA. Thanks to Frank Pfenning for insightful discussions. The anonymous referees are also thanked for their extremely useful comments and criticisms.

References

1. M. Abadi. Protection in Programming-Language Translations. In *Proc. of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 868–883. Springer-Verlag, 1998.
2. M. Abadi. Access Control in a Core Calculus of Dependency. In J. H. Reppy and J. L. Lawall, editors, *Proc. of ICFP'06*, pages 263–273. ACM, 2006.
3. M. Abadi. Logic in Access Control (Tutorial Notes). In *Proc. of FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 145–165. Springer-Verlag, 2009.
4. M. Abadi, M. Burrows, B. W. Lampson, and G. D. Plotkin. A Calculus for Access Control in Distributed Systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, 1993.
5. I. G. Baltopoulos and A. D. Gordon. Secure Compilation of a Multi-Tier Web Language. In *Proc. of TLDI'09*, pages 27–38. ACM, 2009.
6. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement Types for Secure Implementations. In *Proc. of CSF'08*, pages 17–32. IEEE Computer Society, 2008.
7. G. M. Bierman, A. D. Gordon, C. Hritcu, and D. Langworthy. Semantic Subtyping with an SMT Solver. In *Proc. of ICFP'10*, pages 105–116. ACM, 2010.
8. G. M. Bierman, E. Meijer, and W. Schulte. The Essence of Data Access in Cω. In *Proc. of ECOOP'05*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer-Verlag, 2005.
9. L. Caires, J. A. Pérez, J. C. Seco, H. T. Vieira, and L. Ferrão. Type-based Access Control in Data-Centric Systems. Technical Report DIFCTUNL 3/10, U. Nova de Lisboa, 2010.
10. A. Chlipala. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *Proc. of OSDI'10*. USENIX Association, 2010.
11. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *Proc. of FMCO'06*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer-Verlag, 2006.
12. B. J. Corcoran, N. Swamy, and M. W. Hicks. Cross-Tier, Label-Based Security Enforcement for Web Applications. In *SIGMOD Conference 2009*, pages 269–282. ACM, 2009.
13. T. Freeman and F. Pfenning. Refinement Types for ML. In *Proc. of PLDI'91*, pages 268–277. ACM, 1991.

14. D. Garg, L. Bauer, K. D. Bowers, F. Pfenning, and M. K. Reiter. A Linear Logic of Authorization and Knowledge. In *Proc. of ESORICS'06*, volume 4189 of *Lecture Notes in Computer Science*, pages 297–312. Springer-Verlag, 2006.
15. A. D. Gordon and C. Fournet. Principles and Applications of Refinement Types. Technical Report MSR-TR-2009-147, Microsoft Research, 2009.
16. E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD Conference 2006*, pages 706–706. ACM, 2006.
17. N. Swamy, J. Chen, and R. Chugh. Enforcing Stateful Authorization and Information Flow Policies in Fine. In *Proc. of ESOP'10*, volume 6012 of *Lecture Notes in Computer Science*, pages 529–549. Springer-Verlag, 2010.
18. N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A Language for Enforcing User-defined Security Policies. In *Proc. of IEEE S&P'08*, pages 369–383. IEEE Computer Society, 2008.
19. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115:38–94, 1994.

Appendix: Typing Rules for Runtime Expressions

$$\frac{\begin{array}{c}\Delta(t) = \beta_{\{\mathtt{rd}(m_i,R_{m_i})\,|\,m_i\in I\}\cup\overline{\rho}} \qquad J,K\subseteq I \\ \Delta\vdash\bigwedge_{m_j\in J}\theta_{m_j}(R_{m_j}) \qquad \forall_{v_i\in\overline{v}}\ \Delta\vdash v_i:\beta\lfloor_J \\ \forall_{e_i\in\overline{e}}\ \Delta_\pi\vdash e_i:\{b:\mathtt{bool}\,|\,C(v_i,b)\} \qquad \Delta\vdash C(\mathit{this},\mathsf{true})\implies\bigwedge_{m_k\in K}\theta_{m_k}(R_{m_k}) \\ \Delta,x:\beta\lfloor_K,\_:\{\_:\mathtt{unit}\,|\,C(x,\mathsf{true})\}\vdash e_2:\phi\end{array}}{\Delta\vdash \mathsf{from}_t^r\ x\ \mathsf{in}\ \overline{v}\ \mathsf{where}\ \overline{e}\ \mathsf{select}\ e_2:\phi^\star}$$

$$\frac{\begin{array}{c}\Delta(t) = \beta_{\{\mathtt{rd}(m_i,R_{m_i})\,|\,m_i\in I\}\cup\{\mathtt{wr}(m_l,W_{m_l})\,|\,m_l\in L\}\cup\overline{\rho}} \qquad J,K\subseteq I \quad L\subseteq K \\ \Delta\vdash\bigwedge_{m_j\in J}\theta_{m_j}(R_{m_j}) \qquad \forall_{v_i\in\overline{v}}\ \Delta\vdash v_i:\beta \qquad \forall_{v_i'\in\overline{v}'}\ \Delta\vdash v_i':\beta\lfloor_J \\ \forall_{e_i'\in\overline{e}'}\ \Delta_\pi\vdash e_i':\{b:\mathtt{bool}\,|\,C(v_i',b)\} \quad \Delta\vdash C(\mathit{this},\mathsf{true})\implies\bigwedge_{m_k\in K}\theta_{m_k}(R_{m_k}) \\ \beta=\{r:\tau\,|\,I(r)\} \quad I(r)\vdash H(r) \quad H(r)\wedge U(r)\vdash I(r) \quad \phi=\{r:|\tau|_K\,|\,U(r)\} \\ e_i''=e_2\{v_i'/x\} \quad D(y)=(\bigwedge_{m_l\in K-L}y.m_l=x.m_l)\wedge\bigwedge_{m_l\in L}\theta_{m_l}(W_{m_l}\{y/\mathit{this}\}) \\ \Delta,x:\beta\lfloor_K,\_:\{\_:\mathtt{unit}\,|\,C(x,\mathsf{true})\}\vdash e_2:\{y:\phi\mid D(y)\}\end{array}}{\Delta\vdash \mathsf{update}_t^r\ \overline{v}\ \mathsf{with}\ \overline{e'\,?\,e''\,:\,[]}:\mathtt{unit}}$$

$$\frac{\begin{array}{c}\Delta(t) = \beta_{\{\mathtt{rd}(m_k,R_{m_k})\,|\,m_k\in K\}\cup\{\mathtt{wr}(m_l,W_{m_l})\,|\,m_l\in L\}\cup\overline{\rho}} \qquad L\subseteq K \\ \forall_{v_i\in\overline{v}}\ \Delta\vdash v_i:\beta \qquad \Delta\vdash C(\mathit{this},\mathsf{true})\implies\bigwedge_{m_k\in K}\theta_{m_k}(R_{m_k}) \\ \beta=\{r:\tau\,|\,I(r)\} \quad I(r)\vdash H(r) \quad H(r)\wedge U(r)\vdash I(r) \quad \phi=\{r:|\tau|_K\,|\,U(r)\} \\ D(y)(\bigwedge_{m_l\in K-L}y.m_l=x.m_l)\wedge\bigwedge_{m_l\in L}\theta_{m_l}(W_{m_l}\{y/\mathit{this}\}) \\ \forall_{e_i\in\overline{e}}\ (e_i=[])\vee \\ (\Delta\vdash v_i':\beta\lfloor_K\wedge\Delta,\_:\{\_:\mathtt{unit}\,|\,C(v_i',\mathsf{true})\}\vdash e_i:\{y:\phi\mid D(y)\{v_i'/x\}\})\end{array}}{\Delta\vdash \mathsf{update}_t^r\ \overline{v}\ \mathsf{with}\ \overline{e}:\mathtt{unit}}$$

$$\frac{\begin{array}{c}\Delta(t) = \beta_{\{\mathtt{rd}(m_j,R_{m_j})\,|\,m_j\in J\}\cup\{\mathtt{wr}(m_l,W_{m_l})\,|\,l\in L\}\cup\overline{\rho}} \qquad \mathit{fields}(\beta)=L \\ \Delta\vdash\bigwedge_{m_j\in J}\theta_{m_j}(R_{m_j}) \qquad \forall_{v_i\in\overline{v}}\ \Delta\vdash v_i:\beta \qquad \forall_{v_i'\in\overline{v}'}\ \Delta\vdash v_i':\beta\lfloor_J \\ \forall_{e_i\in\overline{e}}\ \Delta_\pi\vdash e_i:\{b:\mathtt{bool}\,|\,C(x,b)\} \qquad \Delta\vdash C(\mathit{this},\mathsf{true})\implies\bigwedge_{m_l\in L}\theta_{m_l}(W_{m_l})\end{array}}{\Delta\vdash \mathsf{delete}_t^r\ \overline{v}\ \mathsf{where}\ \overline{e}:\mathtt{unit}}$$

Fig. 11. Typing Rules (III).