AN ABSTRACT OF THE THESIS OF

Fariba Khan for the degree of Master of Science in Computer Science presented on March 11, 2021.

Title: Formal Verification of the Variational Database Management System

Abstract approved: ______________________________

Eric Walkingshaw

Variation in data is abundant and ubiquitous in real-world applications. Managing variation in databases is, however, difficult and has been extensively studied by the database community. Schema evolution, data integration, and database versioning are examples of well-studied forms of database variation with effective context-specific solutions. However, variation appears in different forms and contexts in databases, and existing approaches cannot be generalized to handle arbitrary forms of variation irrespective of the context. Moreover, in practice, different forms of variation intersect in a particular context. Variational databases (VDB) provide a fundamental solution to variation management by explicitly encoding variation into relational databases that allows addressing different kinds of variation simultaneously. To support expressing variation in information need, traditional relational algebra (RA) is extended to variational relational algebra (VRA). VRA comes with a static type system that checks the validity of variational queries written in VRA. This thesis extends the formalization and formally verifies properties of the variational database management system (VDBMS). Variational sets and set operations definitions are formally verified and VDBs are formally encoded using them. Then, the correctness of the VRA type system with respect to the RA type system is formally specified and verified. VDBMS also allows writing variational queries without repeating variations that are already encoded in the VDB and sub-queries. These implicitly annotated v-queries get explicitly annotated by the system. Therefore, this thesis further formally verifies the process of explicitly annotating variational queries.

Formal Verification of the Variational Database Management System

by

Fariba Khan

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented March 11, 2021
Commencement June 2021

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (Continued)

Chapter 1: Introduction

The difficulty of managing variation is a widely recognized and highly studied problem in the database community. Variation is ubiquitous in nature and society, so it is not surprising that it is ubiquitous in the real-world applications that databases are applied to. However, variation has not been studied as a general concept in databases. Instead, many specific kinds of variation have been addressed by research on schema evolution, data integration, and data versioning. There is a lack of widespread acknowledgment that these are solving different facets of a general problem, potentially with a generic solution. Consequently, variation scenarios that do not fit neatly into one of the scenarios addressed by these well-studied facets must still resort to expensive manual workarounds. Another phenomenon that is common in databases but has not been generally addressed is when different kinds of variation interact. For example, variation can occur in both space and time dimensions in database-backed software produced by software product lines (SPLs) [3, 10, 34, 7]. This thesis extends formalization and formally verifies properties of the Variational Database Management System (VDBMS) [6, 7, 5], which provides a framework for expressing variation in relational databases. The main advantage of VDBMS is that its encoding of variation is generic and explicit, making it suitable for all kinds of variation in relational databases, and for the interactions of multiple kinds of variation. Formal encoding and verification in this thesis are done in *Coq proof assistant* [33].

Variation in databases appears in different levels and dimensions. Databases can vary in the level of their structure (i.e. schema), their content, as well as the queries applied to them. This variation can also happen in two dimensions—time and space. Variation in time refers to changes in the schema, content, or queries of the database over time. Schema evolution and data migration are two well-studied and well-supported forms of database variation in time [27, 11, 4, 32, 29]. Typically in these works, historical changes to the schema are documented in an external document [29] or tracked through timestamps attached to the database [27, 11, 4, 32] so that data can be safely migrated forward towards new versions. They also provide convenient ways to write temporal queries on these temporal representations of databases [21, 32]. On the other hand, variation in "space" refers to the variations that exist in parallel. Work on variation in space exists in the context of data integration [14] which provides a unified interface to query combined data from different sources. Database versioning supported through [8, 19] is another example of variation in space where variation occurs in the level of content and query. Schema evolution, data migration, databases integration and database versioning are specific instances of database variation where variation occurs in some level and dimension. However, different instances of variation can also interact with each other in a specific context.

Database-backed SPLs are an example where database variation in time and space dimensions interact with each other [1]. SPL is an approach to mange variation in software. All supported features of a software system are developed and maintained in a common code base. Variants of the software known as products of the SPL are created by enabling and disabling features based on user requirements. Consequently, databases for products structurally differ from each other in the form of inclusion and exclusion of tables and attributes based on selected features. However, in practice, many SPLs use a single database representation with all tables and features included. Shipping the same database with each product is not efficient, error-prone, and results in lots of null values because of disabled features in individual products [7]. Lots of work has been done to manage structural variation in space in SPLs. One approach is to model data variability in SPLs [2]. The universal data model links SPL features to concepts in the data model and specialized data models can be generated for products. These specialized data models later can be realized as specialized database schemas [22]. Another solution provided in [20] presents a tool that can generate a schema variant for each product from the universal schema by maintaining connection between SPL features and schema elements. However, none of these works consider content-level variation in SPL, nor do they provide support to express information need over databases with structural variations. In addition to variation in space, SPLs also evolve over time as a result of inevitable software evolution. Hence, the shared database needs to evolve over time as well [18]. Variation in time in an SPL database has been separately addressed by adapting existing work on database evolution [18] which, however, cannot encode variation in space.

The partial support for managing variation in SPL databases highlights a gap in research to support the interaction of database variation. In general, in database variation management research, solutions are tailored to specific instances, for example, schema evolution, data versioning, data integration, or model variability for model-oriented instances like SPL. They cannot be generalized to manage any instances of variation or to support variation interaction among different instances. VDBMS fills these gaps in database variation management research.

VDBMS considers variation as an orthogonal concern, which enables encoding different forms of variation directly into the database. Although application-specific solutions might perform better at their niches, VDBMS can be used irrespective of contexts and facilitates interaction among variations that has not been addressed before. In essence, a variational database represents multiple plain relational databases that differ in their structure and content, possibly both in time and space, and encodes the variability explicitly within the database.

Variational databases are defined based on variational sets [16]. This thesis formally encodes variational set and its operations, formally defines and encodes variational set properties, then provides a formal proof of correctness of variational set union and intersection operation. Then, it formally encodes variational databases with the formalized variational sets.

Moreover, VDBMS provides a query language that accounts for variation directly. To support writing variational queries, traditional relational algebra (RA) is extended to variational relational algebra (VRA) by incorporating *choice calculus* [35]. Variational queries written in this extended algebra can express the same intent over different database variants or different intents over different database variants in a variational database. In other words, like variational databases, variational queries also represent multiple plain queries. Due to their extended expressiveness, variational queries are more complicated than plain relational queries. Therefore, VRA is coupled with a static type system to check the validity of variational queries in terms of their compliance with the variation encoded in the schema and content of the targeted variational database.

The VRA type system describes the structure of the result of running a variational query on a variational database. For the VRA type system to be correct, it must preserve variation encoded in variational queries. In other words, the variational result of running a variational query on a variational database must be equivalent to running each variant of the variational query on the corresponding variant of the database. With variation preservation property, VRA type system ensures that each query-database variant is compatible. The variation preservation property along with RA's type safety [30] ensures that VRA is also type safe. This thesis, therefore, provides a formal encoding of VRA and its type system and provides a formal proof of the VRA type system's variation preservation property.

Moreover, to be more user-friendly, VDBMS provides an implicit way of writing variational queries. The idea is that users should not be burdened with including information that can be inferred from the variational schema or from information encoded elsewhere in the query. With this in mind, implicitly annotated v-queries allow not repeating variation already encoded in the variational schema and sub-queries by omitting presence conditions that can be inferred. Users are only required to include additional conditions or constraints they want to impose. These implicitly annotated variational queries get explicitly annotated by the system with variation inferred from the underlying variational schema and sub-queries. Typing of implicitly annotated variational queries, done by the implicit VRA type system, therefore must allow and account for implicitness. To prove the correctness of the implicit VRA type system, it is necessary and sufficient to show that if an implicitly annotated variational query is well-typed in the implicit VRA type system, then its explicitly annotated counterpart is also well-typed in the VRA type system, and that these types are equivalent. This thesis formally encodes the process of explicitly annotating variational queries and provides a formal proof of correctness of the implicit VRA type system with respect to the VRA type system.

The following chapters elaborate the contributions of the thesis. Here is the list of contributions along with the respective chapters and sections that discuss them.

- A formal encoding of variational set and its operations, formal definition and encoding of variational set properties, and a formal proof of correctness of variational set union and intersection operations. (Section 2.3.1).
- A formal encoding of Variational Database (VDB) and its configuration functions that eliminate variation from VDB (Sections 2.3.2 and 2.3.3).
- A formal encoding of Variational Relational Algebra (VRA) and its type system, and the configuration function that eliminates variation from variational queries written in VRA (Section 3.1, 3.2).

- A formal verification of the VRA type system that guarantees the variation preservation property of the type system (Section 3.3).
- A formal encoding of the implicit VRA type system (Section 4.1) and the explicitly annotating function that explicitly annotates the implicitly annotated variational queries(Section 4.2).
- Finally, a formal verification of the implicit VRA type system with respect to the VRA type system (Section 4.3).

Chapter 2: Formal Encoding of Variational Database

A Variational Database (VDB) extends a traditional relational database and represents many variants of a single relational database by capturing where each variant differs from the others. Any database element, therefore, has a conditional presence in the variational database with respect to the variant or variants it belongs to. These presence conditions are encoded explicitly into the database by annotating each element with their respective presence condition.

2.1 Variation Encoding and Variational Elements

Variation is encoded in a database by making its elements' presence conditional. The presence condition of elements is implemented through annotation of elements with *feature expressions* (Section 2.1.1) which are basically boolean expressions. Feature expressions are evaluated to a boolean value through a process called *configuration*. A `true` value indicates that the element is present and a `false` value indicates its absence.

2.1.1 Features and Feature Expressions

Presence conditions need to capture where each variant differs from others in a variational database in terms of which elements are present in a particular variant. Key entities that uniquely identify variants of a database can be used to annotate its elements. For example, in schema evolution, the schema evolves over time and each schema variant can be uniquely identified by a timestamp. Hence, elements present in a variant can be annotated with its respective timestamp. In software product lines (SPL), SPL features create database variants. Hence, features can be used as key identifiers for annotation. In variational databases, we borrow the term *features* from SPL to refer to these key identifiers. For simplicity, features are assumed to be boolean valued variables but can be extended to multi-valued variables. When variants interact with each other, presence conditions of their elements depend on multiple features. These interactions can easily be

Feature Expression Syntax:

$f \in \mathbf{F}$			*Feature Name*
$b \in \mathbf{B}$	$::=$	`true` $\mid$ `false`	*Boolean Value*
$e \in \mathbf{E}$	$::=$	$b \mid f \mid \sim e \mid e \wedge e \mid e \vee e$	*Feature Expression*

Figure 2.1: Feature Expression Syntax.

captured with propositional formulas of features, called feature expressions (Figure 2.1). For example, in schema evolution when an element is present in several schema variants, it can be annotated with disjuncted features representing those variants. In an SPL, if a database element can only be present when two specific SPL features are enabled, then the element can be annotated with the conjunction of two features that represents the two SPL features. In conclusion, features are boolean variables that capture where database variants differ from each other and feature expressions are propositional formulas of features that describe presence conditions of elements in a variational database through annotation.

Feature expressions are formally encoded in Coq as follows.

```
(** Feature Name *)
Definition fname := string.

(** Feature Exression Syntax. *)
Inductive fexp : Type :=
| litB : bool → fexp
| litF : fname → fexp
| comp : fexp → fexp  (* negation *)
| meet : fexp → fexp → fexp (* conjunction *)
| join : fexp → fexp → fexp. (* disjunction *)

Notation "∼(F) f" := (comp f) (at level 35, right associativity).
Infix "∧ (F)" := meet (at level 41, right associativity).
Infix "∨ (F)" := join (at level 45, right associativity).
```

2.1.2 Configuration/Variation Elimination

The set of all features of a variational database, denoted by **F**, represents its configuration space. The configuration space of a variational database is assumed to be closed for

Feature Expression Configuration:

$$\mathbb{E}[\![.]\!] : \mathbf{E} \rightarrow \mathbf{C} \rightarrow \mathbf{B}$$
$$\mathbb{E}[\![b]\!]_c = b$$
$$\mathbb{E}[\![f]\!]_c = c\ f$$
$$\mathbb{E}[\![\sim e]\!]_c = \begin{cases} \texttt{true}, & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{false} \\ \texttt{false}, & \text{otherwise} \end{cases}$$

$$\mathbb{E}[\![e_1 \wedge e_2]\!]_c = \begin{cases} \texttt{true}, & \text{if } \mathbb{E}[\![e_1]\!]_c = \texttt{true} \text{ and } \mathbb{E}[\![e_2]\!]_c = \texttt{true} \\ \texttt{false}, & \text{otherwise} \end{cases}$$
$$\mathbb{E}[\![e_1 \vee e_2]\!]_c = \begin{cases} \texttt{true}, & \text{if } \mathbb{E}[\![e_1]\!]_c = \texttt{true} \text{ or } \mathbb{E}[\![e_2]\!]_c = \texttt{true} \\ \texttt{false}, & \text{otherwise} \end{cases}$$

Figure 2.2: Feature Expression Configuration.

simplicity. A *configuration* c maps each feature in $\mathbf{F}$ to a boolean value. When applied to a variational database, a configuration produces a particular database variant after all variations have been eliminated.

The *feature expression configuration* function evaluates a feature expression with respect to a particular configuration of its pertaining features as defined in Figure 2.2 and encoded as the Coq function `semE` shown below.

```
(** Feature Configuration. *)
Definition config := fname → bool.

(** Feature Expression Configuration. *)
Fixpoint semE (e : fexp) (c : config) : bool :=
match e with
| litB b      ⇒ b
| litF f      ⇒ c f
| ∼(F) e      ⇒ negb (semE e c)
| e1 ∧ (F) e2 ⇒ (semE e1 c) && (semE e2 c)
| e1 ∨ (F) e2 ⇒ (semE e1 c) || (semE e2 c)
end.

Notation "E[[ e ]] c" := (semE e c) (at level 50, left associativity).
```

Feature Expression Properties:

$$\begin{aligned} e_1 \equiv_e e_2 \ \textit{iff} \ & \forall c \in \mathbf{C} : \mathbb{E}[\![e_1]\!]_c = \mathbb{E}[\![e_2]\!]_c \\ \textit{sat}(e) \ \textit{iff} \ & \exists c \in \mathbf{C} : \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \textit{unsat}(e) \ \textit{iff} \ & \forall c \in \mathbf{C} : \mathbb{E}[\![e]\!]_c = \texttt{false} \\ e_1 \rightarrow e_2 \ \textit{iff} \ & \forall c \in \mathbf{C} : \mathbb{E}[\![e_1]\!]_c = \texttt{true} \rightarrow \mathbb{E}[\![e_2]\!]_c = \texttt{true} \end{aligned}$$

Figure 2.3: Feature Expression Properties.

Properties of feature expressions are shown in Figure 2.3. Two feature expressions e_1 and e_2 are equivalent, denoted by $e_1 \equiv_e e_2$, if, for all configurations, they result in the same boolean value. A feature expression e is satisfiable, denoted by $sat(e)$, if there exists a configuration for which it evaluates to true, otherwise it is unsatisfiable, denoted by $unsat(e)$. A feature expression e_1 implies another feature expression e_2, denoted by $e_1 \rightarrow e_2$, if, for any configuration, first expression evaluates to true then, with the same configuration, the later evaluates true as well. Feature expression equivalence, satisfiability, unsatisfiability, and implication are encoded in Coq as `equivE`, `sat`, `unsat` and `implies`, respectively.

```
(** Feature Expression Properties *)

(** Feature Expression Equivalence *)
Definition equivE : relation fexp :=
fun e e' ⇒  forall c, (semE e c) = (semE e' c).
Infix "=e=" := equivE (at level 70) : type_scope.

(* Feature Expression Satisfiability *)
Definition sat (e:fexp): Prop :=
exists c, semE e c = true.

Definition unsat (e:fexp): Prop :=
forall c, semE e c = false.

(* Feature Expression Implication *)
Definition implies (e1 e2:fexp) (c:config) : Prop :=
(E[[ e1 ]] c) = true →  (E[[ e2 ]] c) = true.
Notation "e1 -e→  e2 | c" := (implies e1 e2 c) (at level 91, left
associativity).
```

The set of all possible configurations, denoted by $\mathbf{C}$, represents a set of identifiers for all possible variants of a variational database. Because of interactions among variants, some configurations may be invalid. This can be captured again by a feature expression known as *feature model*. The entire variational database is annotated with the feature model to restrict its configuration space. For example, let's assume, in an SPL S, an SPL product can either be educational, denoted by feature, *edu* or commercial, denoted by, *com*. That is, SPL features *edu* and *com* cannot be `true` at the same time. Since SPL features create database variants in SPLs, the configuration space of the variational database for the SPL S can be restricted with the feature model $(edu \wedge \sim com) \vee (\sim edu \wedge com)$ which enforces that exactly one of these features must be `true` and the other must be `false`.

2.2 Relational Databases

A relational database stores information in a structured way enforced by its schema. A database *schema* S is a finite set of *relation schemas* $\{R_1, ..., R_n\}$. A *relation schema* R is a finite set of attributes $r\{a_1, ..., a_k\}$ where r denotes the name of the relation.

The content of a database is organized under the structure provided by its relation schemas. The respective content for each relation schema is called the *relation content*. A *relation content* RC of a relation schema R is a finite set of *tuples* $\{U_1, ..., U_m\}$. Each tuple $U_i, \forall_{i \in [1,m]}$ contains values $(v_1, ..., v_k)$ corresponding to respective R's set of *attributes* $\{a_1, ..., a_k\}$. The pair of a relation content RC and its relation schema R is called a *table* $T = (R, RC)$. An *instance* I of a database with schema S is a set of tables $\{T_1, ..., T_n\}$.

2.3 Variational Databases

Variational databases extend relational databases to include support for variation. The basic structures underlying elements of a relational database are sets. Hence, Section 2.3.1 extends traditional sets and set operations to variational sets [16]; this section also includes formal proofs of variational set properties (Section 2.3.1.1) and the correctness of variational set operations with respect to traditional set operations (Section 2.3.1.3). Finally, variational schemas and variational tables, that is the structure and contents of variational databases, are defined in Sections 2.3.2 and 2.3.3, respectively.

2.3.1 Variational Set

Variational sets are sets of elements with conditional presence, that is, sets of variational elements (Section 2.1). Each set element is annotated with a feature expression that encodes its presence condition. For example, $X_{v1} = \{{x_1}^{f_1 \wedge f_2}, {x_2}^{f_2 \vee f_3}, {x_3}^{f_3}\}$ is a variational set where elements x_1, x_2, and x_3's presence conditions are denoted by the feature expressions $(f_1 \wedge f_2)$, $(f_2 \vee f_3)$, and f_3 respectively. A variational set itself can be annotated with a feature expression, for example, $X_{v1}^{f} = \{{x_1}^{f_1 \wedge f_2}, {x_2}^{f_2 \vee f_3}, {x_3}^{f_3}\}^{f}$. Annotating a variational set with a feature expression f further restricts the condition under which its variational elements are present. This is equivalent to conjuncting each constituting variational element's annotation with f, for example, $\{{x_1}^{f_1 \wedge f_2}, {x_2}^{f_2 \vee f_3}, {x_3}^{f_3}\}^{f}$ is equivalent to $\{{x_1}^{f_1 \wedge f_2 \wedge f}, {x_2}^{f_2 \vee f_3 \wedge f}, {x_3}^{f_3 \wedge f}\}$. *Variational sets* are alternatively called as *v-sets* and they are used interchangeably in this thesis.

Plain and variational sets are formally encoded in Coq as follows. Plain elements, `elem` are encoded as strings. Variational elements, `velems` are annotated `elems` where annotations are feature expressions, `fexp`.

```
(* Plain Element *)
Definition elem : Type := string.

(* Variational Element *)
Inductive velem : Type :=
| ae : elem → fexp → velem.
```

Plain sets are simply sets of plain elements, `elem` and variational sets are sets of variational elements, `velem`.

```
(* Plain Element Set *)
Definition elems : Type := set elem.

(* Variational Element Set *)
Definition velems : Type := set velem.

(* Annotated Variational Element *)
Definition avelems : Type := (velems * fexp) %type.
```

Conceptually, a variational set represents multiple plain sets which are called variants of the variational set. A plain set is generated from a variational set by configuring feature expressions of its variational elements with respect to a particular configuration and including elements with `true` value. This process is called variational set configuration

V-Set Configuration:

$$\mathbb{X}[\![.]\!] : \mathbf{X_v} \to \mathbf{C} \to \mathbf{X}$$

$$\mathbb{X}[\![\{x^e\} \cup X_v]\!]_c = \begin{cases} \{x\} \cup \mathbb{X}[\![X_v]\!]_c, & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \mathbb{X}[\![X_v]\!]_c, & \text{otherwise} \end{cases}$$

$$\mathbb{X}[\![\{\}]\!]_c = \{\}$$

Figure 2.4: Variational Set Configuration.

Annotated V-Set Configuration:

$$\mathbb{AX}[\![.]\!] : \mathbf{AX_v} \to \mathbf{C} \to \mathbf{X}$$

$$\mathbb{AX}[\![\ {X_v}^e\ ]\!]_c = \begin{cases} \mathbb{X}[\![X_v]\!]_c, & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \{\}, & \text{otherwise} \end{cases}$$

Figure 2.5: Annotated Variational Set Configuration.

(Figure 2.4). For example, $\{f_1, f_2, f_3\}$ is the set of all features in the variational set X_{v1}. Mapping it to {`true`, `false`, `true`} evaluates presence of condition of x_1 to `false`, but presence of conditions of both x_2 and x_3, to `true`. Hence, the corresponding plain set is $\{x_2,\ x_3\}$. Similarly, configuring X_{v1} with feature configuration {`true`, `true`, `false`} generates another plain set, $\{x_1, x_2\}$.

Variational set and annotated variational set configuration shown in Figure 2.4 and 2.5 are encoded in Coq as below.

```
(* Variational Set Configuration X[]c *)
Fixpoint configVElems (ves : velems) (c : config) : elems :=
match ves with
| nil                 ⇒  nil
| cons (ae a e) ves ⇒  if semE e c
                          then (cons a (configVElems ves c))
                          else (        configVElems ves c )
end.
Notation "X[[ ves ]] c" := (configVElems ves c) (at level 50).

(* Annotated Variational Set *)
Definition avelems : Type := (velems * fexp) %type.

(* Annotated Variational Set Configuration AX[]c *)
```

```
Definition configaVElems (aves : avelems) (c : config) : elems :=
match aves with
|(ves, e) ⇒  if semE e c then configVElems ves c else  nil
end.
Notation "AX[[ ves ]] c" := (configaVElems ves c) (at level 50).
```

The set of all possible configurations represents the all possible variants of a variational set. Variational set properties and operations are defined with respect to the respective properties and operations for plain sets.

2.3.1.1 Variational Set Properties

Variational set needs to preserve plain set property of having distinct elements over variation elimination, that is, configuring a variational set should generate a plain set. For example, $X_{v2} = \{{x_1}^{f_1}, {x_2}^{f_2}, {x_2}^{f_3}\}$ is a variational set as plain element x_2 is repeated with different annotations. Configuring X_{v2} by mapping feature set $\{f_1, f_2, f_3\}$ to $\{\texttt{false}, \texttt{true}, \texttt{true}\}$ results in $\{x_2, x_2\}$ which, however, is not a set. Hence, I define the following property, *No-Dup-Elem* to restrict variational set not to have multiple entries of a plain element with different annotations.

Definition 2.3.1 (No-Dup-Elem)**.** X_v *is a* variational set *with property* No-Dup-Elem *if* $\forall x^e \in X_v, (\nexists e^{'}.\ e^{'} \neq e$ *and* $x^{e^{'}} \in X_v)$.

Technically, any variational set with repeated plain elements with different annotation like X_{v2} doesn't violate set properties in a variational set. However, to ensure correctness of variational set operations with respect to plain set operations (Section 2.3.1.3), it is necessary to restrict them with No-Dup-Elem property . Moreover, this property does not limit expressiveness of variational sets. Any variational set that has repeated plain elements with different annotation can be modified to an equivalent variational set with this expected No-Dup-Elem behavior.

To formally encode No-Dup-Elem as an inductive property, I first define another property *In-Elem* that states that a plain element is in a variational set.

Definition 2.3.2 (In-Elem)**.** *A* plain element x *is in a* variational set X_v*, that is,* (In-Elem x X_v) *if* $\exists e.\ x^e \in X_v$.

In-Elem is formally encoded in Coq as a functional property `InElem`.

```
(* In-Elem Property: Plain Element in V-set *)
Function InElem (a:elem) (l:velems) {struct l}: Prop :=
match l with
| []           ⇒  False
| ae x e :: xs ⇒  x = a ∨  InElem a xs
end.
```

Then, No-Dup-Elem property is formalized in Coq as `NoDupElem` using `InElem`. Note that, `In` property in `Coq Standard Library` is different than `InElem` and is only able to recognize if a variational element is in a variational set.

```
(* No-Dup-Elem Property: No Duplicate Plain Element in V-set *)
Inductive NoDupElem : velems → Prop :=
| NoDupElem_nil : NoDupElem nil
| NoDupElem_cons : forall a e l,   InElem a l →  NoDupElem l
→  NoDupElem ((ae a e)::l).
```

To achieve No-Dup-Elem property in a variational set, annotations of all occurrences of an plain element are disjuncted to form a new feature expression. Then, all occurrences of that plain element are replaced by a single entry annotated with the new feature expression. I define the function, *nodup-elem*, that takes any variational set and returns an equivalent variational set that has the No-Dup-Elem property. nodup-elem is defined with three helper functions *existsb-elem*, *get-annot* and *remove-elem*.

Definition 2.3.3 (existsb-elem)**.** *For any plain element x and v-set X_v,* (existsb-elem x X_v) *is* `true` *if plain element a exists in X_v with some annotation, and* `false`*, otherwise.*

Definition 2.3.4 (get-annot)**.** *For any plain element x and v-set X_v,* (get-annot x X_v) *finds all occurrences of x in X_v, concatenates their annotations with boolean* OR *($\vee$) and returns the concatenated annotation.*

Definition 2.3.5 (remove-elem)**.** *For any plain element x and v-set X_v,* (remove-elem x X_v) *removes all occurrences of plain element x in X_v, that is, all variational elements in v-set that have underlying plain element x.*

Definition 2.3.6 (nodup-elem)**.** *For any v-set X_v,* (nodup-elem X_v) *returns a v-set equivalent to X_v and has* No-Dup-Elem *property, that is, it concatenates annotations of each plain element x that exists in X_v (Definition 2.3.3), using* get-annot *in definition 2.3.4 and keeps* one *occurrence of x with the concatenated annotation removing others with* remove-elem *from definition 2.3.5.*

Above functions existsb-elem, get-annot, remove-elem and nodup-elem are encoded in Coq as `existsbElem`, `get_annot`, `removeElem` and `nodupelem` respectively.

```
(* Check whether Plain element a exists in velems \vElemList *)
Definition existsbElem (a : elem) (A : velems) := existsb (eqbElem a) A.

(* Get concatenated annotaion of a Plain element a from velems A *)
Definition get_annot (a : elem) (A: velems) : fexp :=
fold_right Feature.join (litB false) (map (sndVelem) (filter (eqbElem a) A)).

(* Remove all occurances of a Plain element a from velems A *)
Function removeElem (a : elem) (A: velems) {struct A} : velems :=
match A with
| nil ⇒ nil
| ae a' e' :: A' ⇒ match (string_beq a a') with
| true ⇒ removeElem a A'
| false ⇒ ae a' e' :: removeElem a A'
end
end.

(* Concatenation of Duplicate Plain Elements in V-set *)
Function nodupelem (v : velems) {measure List.length v} : velems :=
match v with
| nil          ⇒ nil
| ae a e :: vs ⇒  match existsbElem a vs with
               | false ⇒ ae a e :: nodupelem vs
               | true  ⇒ let e' := get_annot a vs in
               (ae a (e ∨ (F) e') ) :: nodupelem (removeElem a vs)
               end
end.
all:intros; simpl; eauto.
Defined.
```

Following lemma proves that resultant variational set from nodup-elem has No-Dup-Elem property.

Lemma 2.3.7. *For any* v-set*, X_v, No-Dup-Elem (nodup-elem X_v).*

Formal proof of the above lemma encoded as `NoDupElem_nodupelem` in Coq is given below.

```
(* nodupelem ensures NoDupElem  *)
Lemma NoDupElem_nodupelem (v:velems) : NoDupElem (nodupelem v).
Proof. functional induction (nodupelem v) using nodupelem_ind.
+ apply NoDupElem_nil.
+ apply NoDupElem_cons. rewrite InElem_nondupelem.
  rewrite ← existsbElem_InElem.
  rewrite e1. apply diff_false_true. auto.
```

```
+ apply NoDupElem_cons. rewrite InElem_nondupelem.
  apply notInElem_removeElem. apply IHl.
Qed.
```

To prove that function nodup-elem generates an equivalent variational set, we first need to define an equivalence relation on plain lists. Note that, configured variational sets without No-Dup-Elem property generate plain lists, not plain sets. Two plain lists are defined to be equivalent if they cover the same set of elements, irrespective of their order or number of occurrences in the lists.

Definition 2.3.8 (Plain list equivalence)**.** *Two* plain lists*, l_1 and l_2 are equivalent, denoted by $l_1 \equiv_{list} l_2$, iff $\forall x.\ x \in l_1 \iff x \in l_2$.*

Then, two variational sets are defined to be equivalent with respect to plain lists if, for all configurations, their respective configured plain lists are equivalent.

Definition 2.3.9 (V-set equivalence-list)**.** *Two* variational sets*, X_{v1} and X_{v2} are equivalent, denoted by $X_{v1} \equiv_{vlist} X_{v2}$ iff $\forall c.\ \mathbb{X}[\![X_{v1}]\!]_c \equiv_{list} \mathbb{X}[\![X_{v2}]\!]_c$.*

Plain list equivalence and V-set equivalence-list are encoded in Coq as `equiv_elems_list` and `equiv_velems_list`.

```
(* Plain list equivalence *)
Definition equiv_elems_list : relation list elem :=
fun A A' ⇒ forall a, (In a A ↔ In a A').

Infix "=list=" := equiv_elems_list (at level 70) : type_scope.

(* V-set equivalence-list *)
Definition equiv_velems_list : relation velems :=
fun A A' ⇒ forall c, (X[[A]]c) =list= (X[[A']]c).

Infix "=vlist=" := equiv_velems_list (at level 70) : type_scope.
```

Following lemma proves that nodup-elem generates an equivalent variational set.

Lemma 2.3.10 (nodupelem-gen-equiv-velem)**.** *For any variational set X_v, nodupelem $X_v \equiv_{vlist} X_v$.*

Above lemma is encoded in Coq as `nodupelem_gen_equiv_velems_list` as below and its corresponding proof is included in Appendix A.1.1.

```
Lemma nodupelem_gen_equiv_velems_list: forall v, v =vlist= (nodupelem v).
Proof. (See Appndix  A.1.1 ). Qed.
```

From now on, in this thesis, variational sets are assumed to have No-Dup-Elem property, variational set operations maintain No-Dup-Elem property and *V-set equivalence* is re-defined with respect to the *plain set equivalence.*

Definition 2.3.11 (Plain set equivalence)**.** *Two* plain sets*,* X_1 *and* X_2 *are equivalent, denoted by* $X_1 \equiv_{set} X_2$*, iff* $\forall x.\ x \in X_1 \iff x \in X_2$*.*

Definition 2.3.12 (V-set equivalence)**.** *Two* variational sets*,* X_{v1} *and* X_{v2} *are equivalent, denoted by* $X_{v1} \equiv_{vset} X_{v2}$ *iff* $\forall c.\ \mathbb{X}[\![X_{v1}]\!]_c \ \equiv_{set}\ \mathbb{X}[\![X_{v2}]\!]_c$*.*

Definition 2.3.13 (Annotated V-set equivalence)**.** *Two* variational sets*,* ${X_{v1}}^{e_1}$ *and* ${X_{v2}}^{e_2}$ *are equivalent, denoted by* ${X_{v1}}^{e_1} \equiv_{avset} {X_{v2}}^{e_2}$ *iff* $\forall c.\ \mathbb{AX}[\![{X_{v1}}^{e_1}]\!]_c \ \equiv_{set}\ \mathbb{AX}[\![{X_{v2}}^{e_2}]\!]_c$*.*

Plain set equivalence, V-set equivalence and Annotated V-set equivalence are encoded in Coq as `equiv_elems`, `equiv_velems` and `equiv_avelems` respectively.

```
(* Plain set equivalence *)
Definition equiv_elems : relation elems:=
fun A A' ⇒ forall a, (In a A ↔ In a A') ∧
( count_occ string_eq_dec A a = count_occ string_eq_dec A' a).

Infix "=set=" := equiv_elems (at level 70) : type_scope.

(* V-set equivalence *)
Definition equiv_velems : relation velems :=
fun A A' ⇒ forall c, X[[ A]] c =set= X[[ A']]c.

Infix "=vset=" := equiv_velems (at level 70) : type_scope.

(* Annotated V-set equivalence *)
Definition equiv_avelems : relation vqtype :=
fun X X' ⇒ forall c, AX[[ X]]c =set= AX[[ X']]c.

Infix "=avset=" := equiv_avelems (at level 70) : type_scope.
```

Plain set, v-set and annotated v-set equivalences are equivalence relations with reflexivity, symmetry and transitivity. Formal proofs of these properties are included in Appendix A.1.2, A.1.3 and A.1.4. In the rest of the thesis, above equivalence relations are going to be used for plain set, v-set and annotated v-set equivalence.

Variational set subset, (*V-set subset*) extends plain set *subset* property to variational sets.

Definition 2.3.14 (Plain set subset)**.** *A plain set, X_1 is* subset *of another set X_2 iff elements of X_1 are also elements of X_2, that is, $\forall x.\ x \in X_1 \ \rightarrow\ x \in X_2$.*

Definition 2.3.15 (V-set subset)**.** *The v-set X_{v1}* is subset of *the v-set X_{v2}, denoted by $X_{v1} \subseteq X_{v2}$, iff $\forall x^{e_1} \in X_{v1}$, $\exists e_2$. $e_1 \rightarrow e_2$ and $x^{e_2} \in X_{v2}$, i.e., all plain elements in X_{v1} also exist in X_{v2} with more specific presence condition s.t. in a shared configuration, all elements in configured X_{v1} are present in configured X_{v2}. For example, $\{2^{true}, 3^{f_1}\} \subseteq \{2^{true}, 3^{f_1 \vee f_2}, 4^{true}\}$, however, $\{2^{true}, 3^{f_1}\} \not\subseteq \{2^{true}, 3^{f_1 \wedge f_2}\}$ and $\{2^{true}, 3^{f_1}, 4^{true}\} \not\subseteq \{2^{true}, 3^{f_1 \vee f_2}\}$.*

Plain subset and v-set subset is enocoded in Coq as `subset` and `subset_velems_exp` as follows.

```
(* Plain Set Subset *)
Definition subset (A A': elems):= forall x, (In x A → In x A') ∧
(count_occ string_eq_dec A x <= count_occ string_eq_dec A' x).

(* Variational Set Subset *)
Definition subset_velems_exp (A A': velems) :Prop := forall x e c,
In (ae x e) A ∧ ((E[[ e]]c) = true) →
 exists e', In (ae x e') A' ∧  (E[[ e']]c) = true.
```

Correctness of v-set subset property that it correctly extends the plain set subset property to variational sets can be proved by following theorem which states that if a v-set X_{v1} is subset of X_{v2}, then after configuring X_{v1} and X_{v2} with the same configuration c, generated plain sets have subset relationship and it is true for all configurations.

Theorem 2.3.16. *For any two* v-sets, *X_{v1} and X_{v2}, X_{v1}* is subset of *X_{v2} $\iff$ $\forall c$. $\mathbb{X}[\![X_{v1}]\!]_c$* is subset of *$\mathbb{X}[\![X_{v2}]\!]_c$.*

Above theorem is encoded in Coq as `subset_velems_correctness` as below and its formal proof is included in Appendix A.1.5.

```
Theorem subset_velems_correctness A A' (HndpA: NoDupElem A) (HndpA': NoDupElem A'):
   subset_velems_exp A A' ↔ (forall c, subset (X[[ A]]c) (X[[ A']]c)).
Proof. (See Appndix  A.1.5 ). Qed.
```

This correctness theorem basically provides semantic definition of v-set subset property which can easily be extended to annotated v-set subset property.

Definition 2.3.17 (Annotated v-set subset)**.** *For any two* annotated v-sets*,* ${X_{v}}_1^{e_1}$ *and* ${X_{v}}_2^{e_2}$*,* ${X_{v}}_1^{e_1}$ is subset of ${X_{v}}_2^{e_2}$ *iff* $\forall c.\ \mathbb{AX}[\![{X_{v}}_1^{e_1}]\!]_c$ is subset of $\mathbb{AX}[\![{X_{v}}_2^{e_2}]\!]_c$*.*

Annotated v-set subset is encoded in Coq as `subset_avelems` as shown below.

```
Definition subset_avelems ( A A': avelems ) : Prop := forall c,
subset (AX[[ A]]c) (AX[[ A']]c).
```

2.3.1.2 Variational Set Operations

Plain set operations are extended to variational set operations that maintain equivalency to plain set operations over variation elimination. In other words, configuring result of a variational set operation is equivalent to configuring variational sets first and then applying the respective plain set operation on them. This property is defined as follows.

Definition 2.3.18 (V-set operation variation preservation)**.** *A binary* V-set operation $(*)$ *is variation preserving if for any two* V-sets*,* X_{v1} *and* X_{v2}*,* $\forall c.\ \mathbb{X}[\![X_{v1} * X_{v2}]\!]_c \equiv_{set} \mathbb{X}[\![X_{v1}]\!]_c * \mathbb{X}[\![X_{v2}]\!]_c$*.*

Variational set union and intersection, *V-set union and V-set intersection* extend plain *set union and set intersection* for variational sets. The set union of two plain sets, X_1 and X_2, $X_1 \cup X_2$ is a plain set itself that contains all the elements in X_1 and X_2. Elements that are in both sets are included once. V-set union is defined as follows which maintains the No-Dup-Elem property as discussed in Section 2.3.1.1 to ensure variation preservation property defined in Definition 2.3.18. V-set union overloads the $\cup$ notation used for plain set union.

Definition 2.3.19 (V-set union)**.** *The* union *of two v-sets is the union of their elements with the disjunction of presence conditions if an element exists in both v-sets:* $X_{v1} \cup X_{v2} = \{x^{e_1} \mid x^{e_1} \in X_{v1}, \nexists e_2 . x^{e_2} \in X_{v2}\} \cup \{x^{e_2} \mid x^{e_2} \in X_{v2}, \nexists e_1 . x^{e_1} \in X_{v1}\} \cup \{x^{e_1 \vee e_2} \mid x^{e_1} \in X_{v1}, x^{e_2} \in X_{v2}\}$*. For example,* $\{2^{e_1}, 3^{e_1}\} \cup \{2^{e_2}, 4^{e_4}\} = \{2^{e_1 \vee e_2}, 3^{e_1}, 4^{e_4}\}$*.*

Plain set union is formally encoded in Coq as `elems_union` with `set_union` from `Coq Standard Library` which maintains the set property. V-set union is enocded as `velems_union` which is the composition of two operations `nodupelem` and `set_union`. `nodupelem` ensures that No-Dup-Elem property is maintained in the resultant variational set.

```
(* Plain Set Union  *)
Definition elems_union (A A': elems) : elems := set_union string_eq_dec A A'.

(* Variational Set Union  *)
Definition velems_union (A A': velems) : velems := nodupelem (set_union
velem_eq_dec A A').
```

Both plain and v-set union have the identity property described by the following lemmas.

Lemma 2.3.20 (Plain set union nil-r)**.** *For any plain set X, $\{\} \cup X = X$.*

Lemma 2.3.21 (Plain set union nil-l)**.** *For any plain set X, $X \cup \{\} = X$.*

Lemma 2.3.22 (V-set union nil-r)**.** *For any variational set X_v, $\{\} \cup X_v = X_v$.*

Lemma 2.3.23 (V-set union nil-l)**.** *For any variational set X_v, $X_v \cup \{\} \equiv_{va} X_v$.*

Above lemmas, that is, Lemma 2.3.20, 2.3.21 2.3.22 and 2.3.23 are encoded in Coq as `elems_union_nil_r`, `elems_union_nil_l`, `velems_union_nil_r`, `velems_union_nil_l` respectively and their respective proofs are included in Appendix A.2.1 and A.2.2.

```
(* Plain set union nil-r *)
Lemma elems_union_nil_r: forall A, atts_union A [] =set= A.
Proof. (See Appndix  A.2.1 ). Qed.

(* Plain set union nil-l *)
Lemma elems_union_nil_l: forall A (H: NoDup A), elems_union [] A =set= A.
Proof. (See Appndix  A.2.1 ). Qed.

(* V-set union nil-r *)
Lemma velems_union_nil_r : forall A (H: NoDupElem A), velems_union A [] = A.
Proof. (See Appndix  A.2.2 ). Qed.

(* V-set union nil-l *)
Lemma velems_union_nil_l : forall A (H: NoDupElem A), velems_union [] A =vset= A.
Proof. (See Appndix  A.2.2 ). Qed.
```

Similarly, the set intersection of two plain sets is a plain set that only includes elements present in both sets. V-set intersection extends the plain set intersection definition to variational set maintaining both No-Dup-Elem and variation preservation property. V-set intersection overloads the $\cap$ notation used for plain set intersection.

Definition 2.3.24 (V-set intersection)**.** *The* intersection *of two v-sets is a v-set of their shared elements annotated with the conjunction of their presence conditions, that is,*

$X_{v1} \cap X_{v2} = \{x^{e_1 \wedge e_2} \mid x^{e_1} \in X_{v1}, x^{e_2} \in X_{v2}\}$. *For example,* $\{1^{true}, 2^{\neg e_1}, 3^{e_2}\} \cap \{1^{e_2}, 2^{e_2}, 4^{e_2}\} = \{1^{e_2}, 2^{\neg e_1 \wedge e_2}\}$.

Formal encoding of plain set intersection `elems_inter` uses `set_intersection` from `Coq Standard Library` which takes two plain sets and returns the intersected set. Variational set intersection is encoded following the Definition 2.3.24 as `velems_inter` that is shown below.

```
(* Plain Set Intersection *)
Definition elems_inter (A A': elems) : elems := set_inter string_eq_dec A A'.

(* Variational Set Intersection *)
Function velems_inter (A A' : velems) {measure List.length A} : velems
:=
match A with
| nil          ⇒  nil
| ae a e :: As ⇒   match existsbelem a A' with
               | false ⇒  velems_inter As A'
                   | true  ⇒  let e' := get_annot a A' in
                        (ae a (e ∧ (F) e') ) :: velems_inter As A'
                   end
end.
all:intros; simpl; eauto.
Defined.
```

Plain set intersection and v-set intersection have the identity property as shown below.

Lemma 2.3.25 (Plain set intersection nil-r)**.** *For any plain set* X*,* $\{\} \cap X = \{\}$*.*

Lemma 2.3.26 (Plain set intersection nil-l)**.** *For any plain set* X*,* $X \cap \{\} = \{\}$*.*

Lemma 2.3.27 (V-set intersection nil-r)**.** *For any variational set* X_v*,* $\{\} \cap X_v = \{\}$*.*

Lemma 2.3.28 (V-set intersection nil-l)**.** *For any variational set* X_v*,* $X_v \cap \{\} = \{\}$*.*

Lemma 2.3.25, 2.3.26 2.3.27 and 2.3.28 are encoded in Coq as `elems_inter_nil_r`, `elems_inter_nil_l`, `velems_inter_nil_r`, `velems_inter_nil_l` respectively and their respective proofs are included in Appendix A.2.4 and A.2.5.

```
(* Plain set intersection nil-r *)
Lemma elems_inter_nil_r: forall A, elems_inter A [] = [].
Proof. (See Appndix  A.2.4 ). Qed.

(* Plain set intersection nil-l *)
```

```
Lemma elems_inter_nil_l: forall A, elems_inter [] A = [].
Proof. (See Appndix  A.2.4 ). Qed.

(* V-set intersection nil-r *)
Lemma velems_inter_nil_r : forall A, velems_inter A [] = [].
Proof. (See Appndix  A.2.5 ). Qed.

(* V-set intersection nil-l *)
Lemma velems_inter_nil_l : forall A, velems_inter [] A = [].
Proof. (See Appndix  A.2.5 ). Qed.
```

Finally, cross product of v-sets is defined as below. V-set cross product is not formalized in Coq in this thesis. It is not required for any of the VDBMS correctness properties specified and proved in this thesis. However, to ensure correctness of v-set cross product definition, that is, that it correctly extends plain set cross product, future extension of this work should formalize and prove the variation preservation property for v-set cross product.

Definition 2.3.29 (V-set cross product)**.** *The* cross product *of two v-sets is a pair of every two elements of them annotated with the conjunction of their presence conditions.* $X_{v1} \times X_{v2} = \{(x_1, x_2)^{e_1 \wedge e_2} \;\mid\; x_1^{e_1} \in X_1, x_2^{e_2} \in X_2\}$

V-set union and intersection can be extended to annotated v-set with semantic equivalence relationship which requires following two operations to be defined on annotated v-sets.

Add-annot operation allows adding more constraint on an already annotated variational v-set, in other words, annotates an already annotated set, ${X_v}^e$, with another feature expression, e', which is just a syntactic sugar for ${X_v}^{(e \wedge e')}$.

Definition 2.3.30 (Add-Annot)**.** *For any annotated v-set* $Q_v = {X_v}^e$ *and a feature expression* e'*,* (Add-Annot Q_v e')*, denoted by* ${Q_v}^{\wedge\wedge e'}$*, is defined to be* ${X_v}^{(e \wedge e')}$*.*

Add-annot is encoded in Coq as `addannot` as shown below.

```
Definition addannot (Q:vqtype) (e:fexp): vqtype := (fst Q, (snd Q) ∧ (F) e).
Notation " Q ^^ e " := (addannot Q e) (at level 70).
```

Push-annot operation annotates each variational element in a variational set, X_v, with a given feature expression, e.

Definition 2.3.31 (Push-annot)**.** *For a variational set* $X_v = \{{x_{v1}}^{e_1}, ..., {x_{vk}}^{e_k}\}$ *and a feature expression* e*,* (push-annot X_v e)*, denoted by* $(X_v < e)$*, is defined as the v-set* $\{{x_{v1}}^{e_1 \wedge e}, ..., {x_{vk}}^{e_k \wedge e}\}$*.*

Push-annot is encoded in Coq as `push_annot` as shown below.

```
(** Push annotation into a variational element set *)
Fixpoint push_annot (A: velems) (m: fexp) : (velems):=
   match A with
   | nil ⇒ nil
   | ae x e :: xs ⇒  (ae x (e ∧ (F) m)) :: push_annot xs m
   end.
Notation " Q < e " := (push_annot Q e) (at level 70).
```

An annotated v-set can be converted to an equivalent non-annotated v-set using the *push-annot* operation which is later used to extend v-set operations to annotated v-set operations. Following lemma proves that above statement is correct with respect to the v-set equivalence (Definition 2.3.12). Note that, equality itself is an equivalence relation.

Lemma 2.3.32. *Any annotated variational set,* ${X_v}^e$ *is equivalent to the v-set* (push-annot X_v e) *with respect to their respective configured plain sets i.e.* $\forall c,\ \mathbb{AX}[\![{X_v}^e]\!]_c = \mathbb{X}[\![X_v\ <\ e]\!]_c$.

Lemma 2.3.32 is encoded in Coq as `push_annot_correctness` which is shown below along with its formal proof.

```
Lemma push_annot_correctness A e c:
   AX[[ (A, e)]] c = X[[ A < e]] c.
Proof. induction A. simpl.
      destruct ( E[[ e]] c); reflexivity.
      unfold push_annot; fold push_annot.
      destruct a. simpl configVQtype.
      simpl (AX[[_]]c) in IHA.
      simpl (X[[_]]c). simpl (AX[[_]]c).
      destruct (E[[ e]] c); destruct (E[[ f]] c); simpl;
      try(eauto).
      rewrite IHA. reflexivity.
Qed.
```

Now, v-set union is extended to annotated v-set union using push-annot operation.

Definition 2.3.33 (Annotated v-set union)**.** *The* union *of two annotated v-sets* ${X_{v1}}^{e_1}$ *and* ${X_{v2}}^{e_2}$*, denoted by* ${X_{v1}}^{e_1} \cup {X_{v2}}^{e_2}$*, is defined as* $(X_{v1} < e_1) \cup (X_{v2} < e_2)$*.*

Annotated v-set intersection, however, is defined solely in terms of variational set intersection without any helper function.

Definition 2.3.34 (Annotated v-set intersection)**.** *The* intersection *of two annotated v-sets, ${X_{v1}}^{e_1}$ and ${X_{v2}}^{e_2}$, denoted by ${X_{v1}}^{e_1} \cap {X_{v2}}^{e_2}$, is defined as $(X_{v1} \cap X_{v2},\ e_2 \wedge e_2)$ where the later $\cap$ indicates the v-set intersection.*

Note that, annotated v-set union and intersection again overload the $\cup$ and $\cap$ notations. Annotated v-set union and intersection operations are encoded as `avelems_union_vq` and `avelems_inter_vq` respectively.

```
(* Annotated Variational Set Union *)
Definition avelems_union_vq (Q Q': avelems) : avelems :=
let (A, e) := Q in
 let (A', e') := Q' in
  (velems_union (A < e) (A' < e'), e ∨ (F) e').

(* Annotated Variational Set Intersection *)
Definition avelems_inter_vq (Q Q': avelems) : avelems :=
let (A, e) := Q in
 let (A', e') := Q' in
  (velems_inter A A', e ∧ (F) e').
```

Both operations maintain the No-Dup-Elem and the variation preservation property. Corresponding formal proofs are included in the Appendix A.2.7 and A.2.8.

2.3.1.3 Correctness of Variational Set Operations

V-set operations need to be variation preserving (Definition 2.3.18) to ensure that they correctly extend corresponding plain set operations for variational sets. In this section, I provide formal proofs of variation preservation for v-set union and intersection operations defined in Section 2.3.1.2. Following theorem proves v-set union is variation preserving.

Theorem 2.3.35. *For any two* v-sets*, X_v and ${X_v}'$, $\forall c.\ \mathbb{X}[\![X_v \cup {X_v}']\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![{X_v}']\!]_c$.*

Formal encoding of the above theorem is given below and corresponding mathematical and formal proofs are included in the Appendix A.2.3.

```
Theorem velems_union_is_variation_preserving : forall A  A' c (HA: NoDupElem A)
(HA': NoDupElem A'),
          X[[ velems_union A A']]c =set= elems_union (X[[ A]] c) (X[[ A']] c).
Proof. (See Appndix  A.2.3 ). Qed.
```

Similarly, variation preservation property of v-set intersection is guaranteed by the theorem below.

Theorem 2.3.36. *For any two* v-sets*, X_v and $X_v{}'$, $\forall c.$* $\mathbb{X}[\![X_v \cap X_v{}']\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cap \mathbb{X}[\![X_v{}']\!]_c$.

Theorem 2.3.36 is encoded as `velems_intersection_is_variation_preserving` and its formal proof is included in the Appendix A.2.6.

```
Theorem velems_intersection_is_variation_preserving : forall A  A' c (HA: NoDupElem
A)(HA': NoDupElem A'),
         X[[ velems_inter A A']] c = elems_inter (X[[ A]] c) (X[[ A']] c).
Proof. (See Appndix  A.2.6 ). Qed.
```

Now that, we have formalized definitions and properties of v-set and v-set operations, we are ready to formalize the variational schema and content of the variational database.

2.3.2 Variational Schema

Variational schemas extend plain relational database schemas to represent multiple plain database schemas at once. A variational schema (*v-schema*), S_v, is an annotated set of *variational relation schemas* $\{R_1, ..., R_n\}^m$. A variational relation schema (*(v-relation schema)*) is an annotated set of variational attributes (*v-attributes*) preceded by a relation name, $r(A_v)^e$ where $A_v = \{a_{v1}, ..., a_{vk}\}$. Variational attributes are variational elements defined in Section 2.3.1. Annotation of v-schema can be used to restrict v-shcema configuration space to valid or expected configurations only. A v-schema with proper annotation defines all valid schema variants of a variational database where the annotation serves as the feature model of the respective VDB by capturing feature relationships of underlying application.

Formal encoding of v-schema is done in terms of variational set encoding (Section 2.3.1). Plain and variational elements defined and encoded in Section 2.3.1 are used as plain and variational attributes. Plain relation schema `relS` is encoded as plain sets of plain attributes and variational relation schema `vrelS` is encoded as annotated v-sets of variational attributes. Both are accompanied with a relation name `r`, encoded as `string`.

V-Relation Schema Configuration:

$$\mathbb{R}[\![.]\!] : \mathbf{R_v} \to \mathbf{C} \to \mathbf{R}$$

$$\mathbb{R}[\![\{r(A_v)^e\}]\!]_c = \begin{cases} r(\mathbb{X}[\![A_v]\!]_c), & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ r(\{\}), & \text{otherwise} \end{cases}$$

Figure 2.6: Variational Relation Schema(V-Relation Schema) Configuration.

V-Schema Configuration:

$$\mathbb{S}[\![.]\!] : \mathbf{S_v} \to \mathbf{C} \to \mathbf{S}$$

$$\mathbb{S}[\![\{R_{v1}, \ldots, R_{vn}\}^m]\!]_c$$

$$= \begin{cases} \{\mathbb{R}[\![R_{v1}]\!]_c, \ldots, \mathbb{R}[\![R_{vn}]\!]_c\}, & \text{if } \mathbb{E}[\![m]\!]_c = \texttt{true} \\ \{\}, & \text{otherwise} \end{cases}$$

Figure 2.7: Variational Schema(V-Schema) Configuration.

```
(*relation name*)
Definition r : Type := string.

(* Plain Relation Schema *)
Definition relS : Type := (r * elems) % type

(* Variational Relation Schema *)
Definition vrelS : Type := (r * avelems) %type.
```

Plain schema `schema` are then plain sets of relation schemas and variational schemas `vschema` are annotated v-sets of v-relation schemas.

```
(* Plain Schema *)
Definition schema : Type := set relS.

(* Variational Schema *)
Definition vschema : Type := ((set vrelS) * fexp) %type.
```

Configurations of v-relation schemas (Figure 2.6) and v-schemas (Figure 2.7) extend from v-set configuration, (Figure 2.4) and generate plain relation schemas and plain schemas respectively. V-relation schema configuration and V-schema configuration are encoded in Coq as `configVRelS` and `configVSchema` respectively.

```
(* Variational Relation Schema Configuration R[]c *)
```

```
Definition configVRelS (vr : vrelS) (c : config) : relS :=
let r := fst vr in
let VA := fst(snd vr) in
let e := snd (snd vr) in
if E[[ e]]c
 then  (r, (A[[VA]]c))
  else  (r, []).
Notation "R[[ vr ]] c" := (configVRelS vr c) (at level 50).

(* Variational Schema Configuration S[]c *)
Definition configVSchema (vs : vschema) (c : config) : schema :=
let VR := fst vs in
let m := snd vs in
if E[[ m]]c
 then  map (fun vr ⇒  (R[[vr]]c)) VR
  else  [].
Notation "S[[ vs ]] c" := (configVSchema vs c) (at level 50).
```

2.3.3 Variational Database Content

The content of a variational database, that is, the variational database instance is organized under the structure of its v-relation schemas. The pair of a v-relation schema and its respective variational relation content (*v-relation content*) is called a variational table (*v-table*). A v-relation content RC_v of a v-relation schema R_v is a finite set of variational tuples or *v-tuples* $\{U_{v1}, ..., U_{vm}\}$. Each v-tuple $U_{vi}, \forall_{i\in[1,m]}$ is an annotated tuple of variational values $(v_{v1}, ..., v_{vk})^e$ that corresponds to respective v-relation schema R_v's set of v-attributes. A variational value is a plain value annotated with some presence condition. Finally, the variational database instance (*v-instance*) is a set of v-tables corresponding to the respective v-schema.

Plain value `val` and variational value `vval` are encoded in Coq as plain element `elem` and variational element `velem`. Plain tuple `tuple` and variational tuple `vtuple` are encoded as `list` of `val` and `list` of `vval`. plain and variational relation content are encoded as `rcontent` and `vrcontent` which are `set` of `tuple` and `set` of `vtuple` respectively. Finally, plain and variational table and instance are encodes as `table`, `vtable`, `instance` and `vinstance` as shown below.

```
(* Plain Value *)
Definition val : Type := elem.
```

Variational List Configuration:

$$\mathbb{V}[\![.]\!] : \mathbf{V_v} \rightarrow \mathbf{C} \rightarrow \mathbf{V}$$

$$\mathbb{V}[\![(v^e : V_v)]\!]_c = \begin{cases} (v : \mathbb{V}[\![V_v]\!]_c), & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \mathbb{V}[\![V_v]\!]_c, & \text{otherwise} \end{cases}$$

$$\mathbb{V}[\![()]\!]_c = ()$$

Figure 2.8: Variational List Configuration.

```
(* Variational Value *)
Definition vval : Type := velem.

(* Plain Tuple *)
Definition tuple : Type := list val.

(* Variational Tuple *)
Definition vtuple : Type := (list vval * fexp) % type..

(* Plain Relation Content *)
Definition rcontent : Type := set tuple.

(* Variational Relation Content *)
Definition vrcontent : Type := set vtuple.

(* Plain Relation Content *)
Definition table : Type := (relS * rcontent) %type.

(* Variational Relation Content *)
Definition vtable : Type := (vrelS * vrcontent) %type.

(* Plain Instance *)
Definition instance : Type := set table.

(* Variational Relation Content *)
Definition vinstance : Type := set vtable.
```

Variational list configuration is shown in Figure 2.8 and is encoded in Coq as `configVElemList` as below.

```
(* Variational List Configuration V[]c *)
Fixpoint configVElemList (vl : list velem) (c : config) : list elem :=
match vl with
| nil                  ⇒ nil
| cons (ae a e) val ⇒  if semE e c
```

V-Tuple Configuration:

$$\mathbb{U}[\![.]\!] : \mathbf{U_v} \to \mathbf{C} \to \mathbf{U}$$
$$\mathbb{U}[\![{V_v}^{e}]\!]_c = \begin{cases} \mathbb{V}[\![V_v]\!]_c, & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ (), & \text{otherwise} \end{cases}$$

V-Relation Content Configuration:

$$\mathbb{RC}[\![.]\!] : \mathbf{RC_v} \to \mathbf{C} \to \mathbf{RC}$$
$$\mathbb{RC}[\![\{U_{v1}, \ldots, U_{vm}\}]\!]_c = \{\mathbb{U}[\![U_{v1}]\!]_c, \ldots, \mathbb{U}[\![U_{vm}]\!]_c\}$$

V-Table Configuration:

$$\mathbb{T}[\![.]\!] : \mathbf{T_v} \to \mathbf{C} \to \mathbf{T}$$
$$\mathbb{T}[\![(R_v, RC_v)]\!]_c = (\mathbb{R}[\![R_v]\!]_c, \mathbb{RC}[\![RC_v]\!]_c)$$

VDB Instance Configuration:

$$\mathbb{I}[\![.]\!] : \mathcal{I}_{\mathbf{v}} \to \mathbf{C} \to \mathcal{I}$$
$$\mathbb{I}[\![\{T_{v1}, \ldots, T_{vn}\}]\!]_c = \{\mathbb{T}[\![T_{v1}]\!]_c, \ldots, \mathbb{T}[\![T_{vn}]\!]_c\}$$

Figure 2.9: Variational Tuple(V-Tuple), Variational Relation Content(V-Relation Content), Variational Table(V-Table), and Variational Database(VDB) Instance Configurations.

```
                            then (cons a (configVElemList val c))
                            else (        configVElemList val c )
end.
Notation "V[[ vl ]] c" := (configVElemList vl c) (at level 50).
```

Configurations of v-tuples, v-relation contents, v-tables, and VDB instances (Figure 2.9) extend from variational list and variational set configuration. They are encoded in Coq as `configVTuple`, `configVRContent`, `configVTable`, and `configVDBInsatnce`, respectively, listed below.

```
(* V-Tuple  Configuration U[]c *)
Definition configVTuple (vtup : vtuple) (c : config) : tuple :=
let VT := fst vtup in
 let e := snd vtup in
```

```
  if E[[ e]]c
   then  (V[[VT]]c)
    else  [].
Notation "U[[ vu ]] c" := (configVTuple vu c) (at level 50).

(* V-Relation Content Configuration T[]c *)
Definition configVRContent (vrc : vrcontent) (c : config) : rcontent :=
map (fun v ⇒  (U[[v]]c)) vrc.
Notation "T[[ vrc ]] c" := (configVRContent vrc c) (at level 50).

(* (* V-Table Configuration T[]c *)
Definition configVTable (vt : vtable) (c : config) : table :=
let vrs := fst vt in
 let vrc := snd vt in
  (R[[ vrs ]]c, RC[[ vrc ]]c).
Notation "T[[ vt ]] c" := (configVTable vt c) (at level 50). *)

(* VDB Insatnce Configuration I[]c *)
Definition configVDBInsatnce (vins : vinstance) (c : config) :
instance :=
map (fun vt ⇒  ((R[[(fst vt)]]c), (T[[ (snd vt) ]]c))) vins.
Notation "I[[ vt ]] c" := (configVDBInsatnce vt c) (at level 50).
```

Chapter 3: Formal Encoding of Variational Queries

The query language for variational database supports for variation as well. A variational query, (*v-query*) expresses variational intent over a subset of relational database variants represented by the variational database, that is, it can express same intent over several variants or different intents over different variants. In other words, a variational query represents multiple plain queries. To accommodate for variation, traditional relation algebra (RA) is extended with *choices* [35, 15] and variational sets (Section 2.3.1) to define variational relational algebra (VRA) (Section 3.1). VRA is more expressive than RA but it comes with the cost of queries written in VRA, that is, v-queries being more complex. Consequently, checking validity of v-queries is not trivial. Hence, VRA is accompanied with a static type system (Section 3.2) that ensures that v-queries conform to the underlying variational schema and to the variation encoded in the content of the variational database. Formal correctness of VRA type system (Section 3.3) guarantees typing of v-queries itself is correct.

3.1 Variational Relational Algebra(VRA)

VRA allows for variation in queries by incorporating choices and variational sets in traditional RA. Choices are structures that introduce variation by providing multiple alternatives with a selector. Variation is eliminated by evaluating the selector under some configuration and selecting the respective alternative. Feature expressions are used as selectors in the current context. As feature expressions are boolean expressions, using them as selectors limits number of alternatives to two. The first alternative corresponds to the `true` value of selector feature expression and the second one, to `false` value. For example, $e\langle x, y\rangle$ is a choice with two alternatives x and y with the selector feature expression, e. For a given configuration c, if e evaluates to `true`, that is, $\mathbb{E}[\![e]\!]_c = \texttt{true}$, $e\langle x, y\rangle$ is replaced by x, otherwise, it is replaced by y. The result of a query written in VRA is a v-table, that is, a pair of v-relation schema and respective v-relation content. The type of a valid v-query generated by VRA type system, discussed in Section 3.2,

V-Condition Syntax:

$$\begin{array}{rcl} \theta_v \in \mathbf{\Theta_v} & := & b \;\mid\; a \bullet k \;\mid\; a \bullet a \;\mid\; \sim \theta_v \;\mid\; \theta_v \vee \theta_v \\ & \mid & \theta_v \wedge \theta_v \;\mid\; e\langle \theta_v, \theta_v \rangle \end{array}$$

VRA Syntax:

$$\begin{array}{rcll} q_v \in \mathbf{Q_v} & := & r & \textit{Variational Relation} \\ & \mid & \sigma_{\theta_v} q_v & \textit{Variational Selection} \\ & \mid & \pi_{{A_v}^e} q_v & \textit{Variational Projection} \\ & \mid & e\langle q_v, q_v \rangle & \textit{Choice of Queries} \\ & \mid & q_v \times q_v & \textit{Variational Cartesian Product} \\ & \mid & q_v \circ q_v & \textit{Variational Set Operation} \\ & \mid & \varepsilon & \textit{Empty Relation} \end{array}$$

Figure 3.1: Variational Relational Algebra(VRA) Definition. $\bullet$ and $\circ$ denote comparison $(<, \leq, =, \neq, >, \geq)$ and v-set operations $(\cap, \cup)$, respectively. b represents boolean values, a denotes plain attributes and k denotes constants.

is a v-relation schema. Running a v-query on a variational database generates a set of v-tuples, that is, the v-relation content that corresponds to its type. The result is formed by combining them into a v-table.

VRA extends traditional operations in RA to support variation, and includes an additional operation, called *choice* as well as an empty relation as shown in VRA syntax in Figure 3.1. The variational selection (v-select) operation extends plain selection operation with variational conditions (Figure 3.1). Variational conditions (v-condition) are plain conditions extended with choice structure. In v-condition syntax, $\bullet$ denotes comparison $(<, \leq, =, \neq, >, \geq)$ and b, a, k represents boolean values, plain attributes and constants, respectively. For example, the query $\sigma_{e\langle a_1 = a_2, a_1 = a_3 \rangle} r$ selects v-tuples from r that meet the condition, $a_1 = a_2$ and annotates them with e, and also, selects v-tuples that satisfies $a_1 = a_3$ and annotates them with $(\sim e)$. Note that, v-tuples are annotated variational set. Annotating an already annotated set with a feature expression is done by the add-annot operation defined in Definition 2.3.30. The variational projection operation takes an annotated v-set of attributes as its parameter and projects attributes present in the parameter with appropriate annotation. For example,the query $\pi_{\{a^{e_1}\}^{e_2}}\ r$ projects a from relation r and annotate the projected v-tuples with $e_1 \wedge e_2$. The choice operation, $e\langle q_{v1}, q_{v2}\rangle$ combines two v-queries q_{v1} and q_{v2} with a feature expression e. Resulted v-tuples from q_{v1} are annotated with e and those from q_{v2}, with $(\sim e)$. In practice, sometimes it is useful to have a choice of v-queries where one alternative does nothing. To support this, VRA is augmented with an empty relation which generates an empty v-query. The rest VRA operations are variational set operations that extend respective plain set operations in RA as defined and discussed in Section 2.3.1.2. In VRA syntax, $\circ$ denotes v-set union ($\cap$) and intersection ($\cup$) operations. The resulted set of v-tuple from running any v-query on a variational database is paired with the v-query's type to form the resultant v-table. Note that, presence conditions of elements in the result of a v-query is at least as specific as their respective presence conditions in the underlying variational database.

Formal encoding of both plain and v-query is given below. Plain and variational condition are encoded in Coq as `cond` and `vcond`. `op` describes the comparison operations and `bool`, `elem`, `nat` denotes the boolean values, plain attributes and constants, respectively. Plain and variational query are encoded as `query` and `vquery`. `setop` denotes the v-set union and intersection operations.

```
Inductive op : Type :=
| eq | GTE | LTE | GT | LT | neq.

(* Plain Condition *)
Inductive cond : Type :=
| litCB  : bool → cond
| elemOpV : op → elem → nat → cond
| elemOpA : op → elem → elem → cond
| negC   : cond → cond
| conjC  : cond → cond → cond
| disjC  : cond → cond → cond.

(* Varitational condition *)
Inductive vcond : Type :=
| litCB_v  : bool → vcond
| elemOpV_v : op → elem → nat → vcond
| elemOpA_v : op → elem → elem → vcond
| negC_v   : vcond → vcond
| conjC_v  : vcond → vcond → vcond
| disjC_v  : vcond → vcond → vcond
| chcC     : fexp → vcond → vcond → vcond.

Inductive setop : Type := union | inter.
```

```
(* Plain Query*)
Inductive query : Type :=
| rel   : relS    → query
| proj  : elems    → query → query
| sel   : cond    → query → query
| prod  : query   → query → query
| setU  : setop   → query → query → query
| empty : query.

(* Variaitonal Query *)
Inductive vquery : Type :=
| rel_v   : vrelS    → vquery
| proj_v  : avelems   → vquery → vquery
| sel_v   : vcond    → vquery → vquery
| chcQ    : fexp     → vquery → vquery → vquery
| prod_v  : vquery   → vquery → vquery
| setU_v  : setop    → vquery → vquery → vquery
| empty_v : vquery.
```

Variational condition configuration ($\mathbb{C}[\![.]\!]_c$) is shown in Figure 3.2. V-query configuration ($\mathbb{Q}[\![.]\!]_c$), also in Figure 3.2, is performed with the help of feature expression configuration ($\mathbb{E}[\![.]\!]_c$), annotated v-set configuration ($\mathbb{AX}[\![.]\!]_c$), v-relation schema configuration ($\mathbb{R}[\![.]\!]_c$), and v-condition configuration ($\mathbb{C}[\![.]\!]_c$), which are defined in Figures 2.2, 2.5, 2.6, and 3.2, respectively. Variational condition and variational query configurations are encoded in Coq as `configVQuery` and `configVCond`, denoted by `(Q[[.]]c)` and `(C[[.]]c)`, respectively.

```
(*Variational Query Configuration Q[]c *)
Fixpoint configVQuery (vq : vquery) (c : config) : query :=
  match vq with
   | rel_v  vr            ⇒ rel (R[[ vr]]c)
   | proj_v avelems vq    ⇒ proj (AX[[avelems]]c) (configVQuery vq c)
   | sel_v  vc  vq        ⇒ sel (C[[ vc]]c) (configVQuery vq c)
   | chcQ e vq1 vq2       ⇒ if E[[ e]]c then configVQuery vq1 c
                                        else configVQuery vq2 c
   | prod_v vq1 vq2       ⇒ prod (configVQuery vq1 c) (configVQuery vq2 c)
   | setU_v setop vq1 vq2 ⇒ setU setop (configVQuery vq1 c) (configVQuery vq2 c)
   | empty_v              ⇒ empty
  end.
Notation "Q[[ vq ]] c" := (configVQuery vq c) (at level 50).

(*  Variational Condition Configuration C[]c *)
Fixpoint configVCond (vc : vcond) (c : config) : cond :=
 match vc with
 | litCB_v  b           ⇒  litCB b
```

V-Condition Configuration:

$$\mathbb{C}[\![.]\!] : \mathbf{\Theta_v} \rightarrow \mathbf{C} \rightarrow \underline{\mathbf{\Theta}}$$
$$\mathbb{C}[\![b]\!]_c = b$$
$$\mathbb{C}[\![\underline{a} \bullet k]\!]_c = \underline{a} \bullet k$$
$$\mathbb{C}[\![\underline{a}_1 \bullet \underline{a}_2]\!]_c = \underline{a}_1 \bullet \underline{a}_2$$
$$\mathbb{C}[\![\sim \theta_v]\!]_c = \sim \mathbb{C}[\![\theta_v]\!]_c$$
$$\mathbb{C}[\![\theta_{v1} \vee \theta_{v2}]\!]_c = \mathbb{C}[\![\theta_{v1}]\!]_c \vee \mathbb{C}[\![\theta_{v2}]\!]_c$$
$$\mathbb{C}[\![\theta_{v1} \wedge \theta_{v2}]\!]_c = \mathbb{C}[\![\theta_{v1}]\!]_c \wedge \mathbb{C}[\![\theta_{v2}]\!]_c$$
$$\mathbb{C}[\![e\langle\theta_{v1}, \theta_{v2}\rangle]\!]_c = \begin{cases} \mathbb{C}[\![\theta_{v1}]\!]_c, & \text{if } \mathbb{E}[\![e]\!]_c \\ \mathbb{C}[\![\theta_{v2}]\!]_c, & \text{otherwise} \end{cases}$$

VRA Configuration:

$$\mathbb{Q}[\![.]\!] : \mathbf{Q_v} \rightarrow \mathbf{C} \rightarrow \mathbf{Q}$$
$$\mathbb{Q}[\![r]\!]_c = \mathbb{R}[\![r]\!]_c = \underline{r}$$
$$\mathbb{Q}[\![\pi_{{A_v}^e} q_v]\!]_c = \pi_{\mathbb{AX}[\![{A_v}^e]\!]_c} \mathbb{Q}[\![q_v]\!]_c$$
$$\mathbb{Q}[\![\sigma_{\theta_v} q_v]\!]_c = \sigma_{\mathbb{C}[\![\theta_v]\!]_c} \mathbb{Q}[\![q_v]\!]_c$$
$$\mathbb{Q}[\![e\langle q_{v1}, q_{v2}\rangle]\!]_c = \begin{cases} \mathbb{Q}[\![q_{v1}]\!]_c, & \text{if } \mathbb{E}[\![e]\!]_c \\ \mathbb{Q}[\![q_{v2}]\!]_c, & \text{otherwise} \end{cases}$$
$$\mathbb{Q}[\![q_{v1} \times q_{v2}]\!]_c = \mathbb{Q}[\![q_{v1}]\!]_c \times \mathbb{Q}[\![q_{v2}]\!]_c$$
$$\mathbb{Q}[\![q_{v1} \circ q_{v2}]\!]_c = \mathbb{Q}[\![q_{v1}]\!]_c \circ \mathbb{Q}[\![q_{v2}]\!]_c$$
$$\mathbb{Q}[\![\varepsilon]\!]_c = \underline{\varepsilon}$$

Figure 3.2: Variational Condition(V-Condition) and Variational Relational Algebra(VRA) Configuration. V-condition and v-query are assumed to be well-typed by the configuration functions.

```
 | elemOpV_v o   a   k  ⇒  elemOpV o a k
 | elemOpA_v  o  a1  a2 ⇒  elemOpA o a1 a2
 | negC_v  vc           ⇒  negC (configVCond vc  c)
 | conjC_v  vc1 vc2     ⇒  conjC (configVCond vc1 c) (configVCond vc2 c)
 | disjC_v  vc1 vc2     ⇒  disjC (configVCond vc1 c) (configVCond vc2 c)
 | chcC e   vc1 vc2     ⇒  if semE e c then configVCond vc1 c
 else configVCond vc2 c
end.
Notation "C[[ vc ]] c" := (configVCond vc c) (at level 70).
```

3.2 VRA Type System

VRA comes with a type system that statically checks if a v-query complies with the underlying variational database. For example, let's assume that we have a variational database with v-schema $S_{v3} = \{r\left({a_1}^{e_1}, {a_2}^{true}\right)^{e_2}\}^{true}$. The $\pi_{\{{a_4}^{true}\}^{true}}\ r$ is not a valid query for S_{v3} as its relation r does not have an attribute a_4. The $\pi_{\{{a_1}^{\sim e_1}\}^{true}}\ r$ is also not a valid query S_{v3} as it intents to project a_1 from r for configurations c that $\mathbb{E}[\![\sim e_1]\!]_c = \texttt{true}$. But a_1 is not present in the relation r under these configurations. However, $\pi_{\{{a_1}^{e_1 \wedge e_2 \wedge e_3}\}^{true}}\ r$ is valid as a_1 is present in r under the configurations c, $\mathbb{E}[\![e_1 \wedge e_2 \wedge e_3]\!]_c = \texttt{true}$. To be more explicit, a_1 is present in the relation r for all configurations c that $\mathbb{E}[\![e_1 \wedge e_2]\!]_c = \texttt{true}$ and for all configurations c, $\mathbb{E}[\![e_1 \wedge e_2 \wedge e_3]\!]_c = \texttt{true} \rightarrow \mathbb{E}[\![e_1 \wedge e_2]\!]_c = \texttt{true}$. Presence conditions of any attribute, a in Q_v of v-query $\pi_{Q_v} q_v$ need to be at least as specific as a's presence condition in the result of q_v.

Type of a v-query is a v-relation schema, $result(A_v)^e$ where $result$ is the relation name which is fixed for all v-queries. The annotated set of v-attributes $({A_v}^e)$ describes which v-attributes are present in the result of the v-query. As relation name is fixed for all v-queries' type, for brevity, types are considered to be annotated set of v-attributes that are basically annotated v-set. Types of plain queries written in traditional RA are

V-Query Type Configuration:

$$\mathbb{QT}[\![.]\!] : \mathbf{QT_v} \rightarrow \mathbf{C} \rightarrow \mathbf{QT}$$
$$\mathbb{QT}[\![QT_v]\!]_c = \mathbb{AX}[\![QT_v]\!]_c$$

Figure 3.3: Variational Query(V-Query) Type Configuration.

sets of plain attribute. Plain and variational query type are encoded in Coq as `qtype` and `vqtype`, respectively. Variational query type (v-type) configuration(Figure 3.3) is done with annotated variational set configuration (Figure 2.5) and is enocoded in as `configVQtype`.

```
(* Plain Query Type *)
Definition qtype  : Type := (elems) %type.

(* Variaitonal Query Type *)
Definition vqtype : Type := avelems.

(* Variational Query Type Configuration QT[]c *)
Definition configVQtype (vqt : vqtype) (c : config) : qtype := AX[[ vqt]]c.
```

Annotated v-set equivalence, subset property (Definitions 2.3.13, 2.3.17) as well as its union and intersection operations (Definitions 2.3.33, 2.3.34) are renamed for v-query type. V-query type equivalence, subset, union and intersection are denoted by $\equiv_{vqtype}$, $\subseteq$, $\cup$ and $\cap$, respectively.

Definition 3.2.1 (V-query type equivalence)**.** *:= Annotated v-set equivalence*

Definition 3.2.2 (V-query type subset)**.** *:= Annotated v-set subset*

Definition 3.2.3 (V-query type union)**.** *:= Annotated v-set union*

Definition 3.2.4 (V-query type intersection)**.** *:= Annotated v-set intersection*

V-query type equivalence, subset, union and intersection operations are encoded in Coq as `equiv_vqtype`, `subset_vqtype`, `vqtype_union_vq` and `vqtype_inter_vq`.

```
(* V-Query Type Equivalence *)
Definition equiv_vqtype : relation vqtype := fun X X' ⇒  X =avset= X'.
Infix "=vqtype=" := equiv_vqtype (at level 70) : type_scope.

(* V-Query Type subset *)
Definition subset_vqtype ( A A': vqtype ) : Prop := subset_avelems.

(* V-Query Type Union *)
Definition vqtype_union_vq (Q Q': vqtype) : vqtype := avelems_union_vq.

 (* V-Query Type Intersection *)
Definition vqtype_inter_vq (Q Q': vqtype) : vqtype := avelems_inter_vq.
```

V-Query Typing Rules:

EmptyRelation-E

$$e, S_v \vDash \varepsilon : \{\}^{\texttt{false}}$$

Relation-E

$$\frac{r(A_v)^{e_r} \in S_v \qquad e' = e_r \wedge pc(S_v) \qquad sat\big(e \wedge e'\big)}{e, S_v \vDash r : {A_v}^{e \wedge e'}}$$

Project-E

$$\frac{e, S_v \vDash q_v : {A_v}'^{e'} \qquad {Q_v}^{\hat{}\,\hat{}\,e} \subseteq {A_v}'^{e'}}{e, S_v \vDash \pi_{Q_v}\ q_v : {Q_v}^{\hat{}\,\hat{}\,e}}$$

Select-E

$$\frac{e, S_v \vDash q_v : {A_v}^{e'} \qquad e, {A_v}^{e'} \vDash \theta_v}{e, S_v \vDash \sigma_{\theta_v}\ q_v : {A_v}^{e'}}$$

Choice-E

$$\frac{e \wedge e', S_v \vDash q_{v1} : {A_{v1}}^{e_1} \qquad e \wedge \sim e', S_v \vDash q_{v2} : {A_{v2}}^{e_2}}{e, S_v \vDash e\langle q_{v1}, q_{v2}\rangle : {A_{v1}}^{e_1} \cup {A_{v2}}^{e_2}}$$

Product-E

$$\frac{e, S_v \vDash q_{v1} : {A_{v1}}^{e_1} \qquad e, S_v \vDash q_{v2} : {A_{v2}}^{e_2} \qquad {A_{v1}}^{e_1} \cap {A_{v2}}^{e_2} = \{\}}{e, S_v \vDash q_{v1} \times q_{v2} : {A_{v1}}^{e_1} \cup {A_{v2}}^{e_2}}$$

SetOp-E

$$\frac{e, S_v \vDash q_{v1} : {A_{v1}}^{e_1} \qquad e, S_v \vDash q_{v2} : {A_{v2}}^{e_2} \qquad {A_{v1}}^{e_1} \equiv_T {A_{v2}}^{e_2}}{e, S_v \vDash q_{v1} \times q_{v2} : {A_{v1}}^{e_1}}$$

Figure 3.4: Variational Relational Algebra(VRA) Typing Relation. The typing rule of a join query is the combination of rules Select-E and Product-E.

V-Condition Typing Rules (b: boolean tag, a: plain attribute, k: constant value):

$$\textsc{Boolean-C}\quad e, A_v \vdash b$$

$$\textsc{AttOptVal-C}\quad \frac{a^{e'} \in A_v \qquad k \in dom_{\mathcal{I}}(a)}{e, A_v \vdash a \bullet k}$$

$$\textsc{AttOptAtt-C}\quad \frac{{a_1}^{e_1} \in A_v \qquad {a_2}^{e_2} \in A_v \qquad type(a_1) = type(a_2)}{e, A_v \vdash a_1 \bullet a_2}$$

$$\textsc{Neg-C}\quad \frac{e, A_v \vdash \theta_v}{e, A_v \vdash \sim \theta_v}$$

$$\textsc{Conjunction-C}\quad \frac{e, A_v \vdash \theta_{v1} \qquad e, A_v \vdash \theta_{v2}}{e, A_v \vdash \theta_{v1} \wedge \theta_{v2}}$$

$$\textsc{Disjunction-C}\quad \frac{e, A_v \vdash \theta_{v1} \qquad e, A_v \vdash \theta_{v2}}{e, A_v \vdash \theta_{v1} \vee \theta_{v2}}$$

$$\textsc{Choice-C}\quad \frac{e \wedge e', A_v \vdash \theta_{v1} \qquad e \wedge \sim e', A_v \vdash \theta_{v2}}{e, A_v \vdash e'\langle \theta_{v1}, \theta_{v2} \rangle}$$

Figure 3.5: Variational Condition(V-Condition) Typing Relation.

Typing relations of VRA are defined as a set of inference rule as in Figure 3.4. A typing relation $e, S_v \vDash q_v : {A_v}^{e'}$ states that in variational context e with underlying v-schema S_v, v-query q_v has type ${A_v}^{e'}$. If a v-query doesn't have a type, it is not a valid query. Type system keeps track of the variation encoded in v-queries with the variational context, e.

The Relation-E rule states that in a variational context e with v-schema S_v with feature model $pc(S_v)$, a relation $r(A_v)^{e_r}$ that is contained in S_v has type ${A_v}^{(e' \wedge e)}$where $e' = e_r \wedge pc(S_v)$ and $(e \wedge e')$ is statisfiable. Note that, $pc(S_v)$ returns the feature model, that is, the presence condition of the variational schema S_v.

The Project-E rule states that, in a variational context, e with v-schema S_v, assuming a v-query q_v has type ${A_v'}^{e'}$, the projection v-query $\pi_{Q_v}\ q_v$ has type ${Q_v}^{\hat{}\hat{}e}$ given that ${Q_v}^{\hat{}\hat{}e} \subseteq {A_v'}^{e'}$. Remember that, variational projection query's parameter, Q_v is an annotated v-set, $\hat{}\hat{}$ is the add-annot operation (Definition 2.3.30) that annotates an already annotated set, and $\subseteq$ is the annotated v-set subset operation (Definition 2.3.17). The subset condition ${Q_v}^{\hat{}\hat{}e} \subseteq {A_v'}^{e'}$ in the rule ensures that all plain attributes in Q_v are present in ${A_v}'^{e'}$ with such presence conditions that in each query variant of $\pi_{Q_v}\ q_v$, to be projected plain attribute set has the subset relationship with the set it is projecting from.

The Select-E rule states that, in a variational context e with v-schema S_v, assuming a v-query q_v has type ${A_v}^{e'}$, the selection v-query $\sigma_{\theta_v}\ q_v$ has the same type, given that, θ_v is well-formed in the same variational context e with respect to ${A_v}^{e'}$, that is, $e, {A_v}^{e'} \vDash \theta_v$. A v-condition θ_v is well-formed in a variational context e with respect to a v-set of attributes A_v, that is, $e, A_v \vDash \theta_v$ if and only if θ_v has the expected syntax and the plain attributes present in the θ_v are also present in A_v.

The rule Choice-E states that, in a variational context e with v-schema S_v, the type of choice of two v-queries q_{v1} and q_{v2}, denoted by $e\langle q_{v1}, q_{v2}\rangle$, is the union of q_{v1} and q_{v2}'s types in the variational contexts $(e \wedge e')$ and $(e \wedge \sim e')$, respectively, with the same v-schema S_v, given that, q_{v1} and q_{v2} are valid in these contexts. In variational contexts for q_{v1} and q_{v2}, e is conjuncted with e' and $\sim e'$ respectively as, in the choice structure, q_{v1} is selected when $\mathbb{E}[\![e']\!]_c = \texttt{true}$ and q_{v2} is selected when $\mathbb{E}[\![e']\!]_c = \texttt{false}$.

The rule Product-E sates that, in a variational context, e with v-schema S_v, the type of the product of two v-queries q_{v1} and q_{v2}, denoted by $(q_{v1} \times q_{v2})$, is the union of the types of q_{v1} and q_{v2} (Definition 3.2.3) in the same context with the same schema. q_{v1} and q_{v2}'s types are needed to be disjoint.

Finally, the SetOp-E rule defines typing rule for v-set operation queries which states that, in a variational context e with v-schema S_v, assuming types of two v-queries q_{v1} and q_{v2} to be ${A_{v1}}^{e_1}$ and ${A_{v2}}^{e_2}$ respectively, type of any v-set operation query among them, denoted by $(q_{v1} \circ q_{v2})$ is ${A_{v1}}^{e_1}$, given that, ${A_{v1}}^{e_1} \equiv_{vqtype} {A_{v2}}^{e_2}$.

VRA type system is encoded in Coq as inductive proposition, `vtype` which is included below.

```
(* ---------------------------------------------------------------
  | Type of variational query ( |= )
   ---------------------------------------------------------------*)

Inductive vtype :fexp → vschema → vquery → vqtype → Prop :=
 (*   -- EMPTYRELATION-E --  *)
 | EmptyRelation_vE : forall e S {HndpRS:NoDupRn (fst S)}
                          {HndpAS: NODupElemRs S},
   vtype e S (empty_v) ([], litB false)

 (*   -- RELATION-E  --  *)
```

```
| Relation_vE : forall e S {HndpRS:NoDupRn (fst S)} {HndpAS:NODupElemRs S}
                        rn {Hrn: empRelInempS rn}A {HA: NoDupElem A} e',
   InVR (rn, (A, e')) S →
    sat (e ∧ (F) e') →
     vtype e S (rel_v (rn, (A, e' ))) (A, (e ∧ (F) e'))

(*   -- PROJECT-E --  *)
| Project_vE: forall e S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                      vq {HndpvQ: NoDupElemvQ vq} e' A' {HndpAA': NoDupElem A'}
                      Q {HndpQ: NoDupElem (fst Q)},
   vtype e S vq (A', e') →
    subset_vqtype (Q^^e) (A', e') →
     vtype e S (proj_v Q vq) (Q^^e)

(*  -- SELECT-E --  *)
| Select_vE: forall e S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                vq {HndpvQ: NoDupElemvQ vq} A {HndpAA: NoDupElem A} e' vc,
  vtype e S vq (A, e') →
   { e, (A, e') |- vc } →
    vtype e S (sel_v vc vq) (A, e')

(*  -- CHOICE-E --  *)
 | Choice_vE: forall e e' S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                       vq1 {HndpvQ1: NoDupElemvQ vq1} vq2 {HndpvQ2: NoDupElemvQ
                       vq2} A1 {HndpAA1: NoDupElem A1} e1 A2 {HndpAA2: NoDupElem
                       A2} e2,
   vtype (e ∧ (F) e') S vq1 (A1, e1) →
    vtype (e ∧ (F) ( (F) e')) S vq2 (A2, e2) →
     vtype e S (chcQ e' vq1 vq2) (vqtype_union_vq (A1, e1) (A2, e2))

(*  -- PRODUCT-E --  *)
| Product_vE: forall e S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                     vq1 {HndpvQ1: NoDupElemvQ vq1} vq2 {HndpvQ2:
                         NoDupElemvQ vq2} A1 {HndpAA1: NoDupElem A1} e1 A2
                         {HndpAA2: NoDupElem A2} e2 ,
   vtype e  S vq1 (A1, e1) →
    vtype e  S vq2 (A2, e2) →
     vqtype_inter_vq (A1, e1) (A2, e2) =vqtype= (nil, litB false) →
      vtype e S (prod_v vq1 vq2) (vqtype_union_vq (A1, e1) (A2, e2))

(*  -- SETOP-E --  *)
| SetOp_vE: forall e S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                   vq1 {HndpvQ1: NoDupElemvQ vq1} vq2 {HndpvQ2: NoDupElemvQ vq2}
                    A1 {HndpAA1: NoDupElem A1} e1 A2 {HndpAA2: NoDupElem A2}  e2
                    op,
   vtype e S vq1 (A1, e1) →
    vtype e S vq2 (A2, e2) →
```

```
      equiv_vqtype (A1, e1) (A2, e2) →
       vtype e S (setU_v op vq1 vq2) (A1, e1).

Notation "{ e , S |= vq | vt }" := (vtype e S vq vt) (e at level 200).

(*-------------------------------------------------------------
 | Type of (|-c ) variational condition
 -------------------------------------------------------------*)

Inductive vcondtype :fexp →  vqtype →  vcond →  Prop :=
 | litCB_vC : forall e Q b,
     vcondtype e Q (litCB_v b)

 | elemOpV_vC : forall e Q o a k,
   (exists e : fexp, In (ae a e) ((fst Q) < (snd Q)) ∧  sat(e)) →
    vcondtype e Q (elemOpV_v o a k)

 | elemOpA_vC : forall e Q o a1 a2,
     (exists e1 : fexp, In (ae a1 e1) ((fst Q) < (snd Q)) ∧  sat(e1)) →
      (exists e2 : fexp, In (ae a2 e2) ((fst Q) < (snd Q)) ∧  sat(e2)) →
       vcondtype e Q (elemOpA_v o a1 a2)

 | NegC_vC : forall e Q c,
    vcondtype e Q c →
     vcondtype e Q (negC_v c)

 | ConjC_vC : forall e Q c1 c2,
    vcondtype e Q c1 →
     vcondtype e Q c2 →
      vcondtype e Q (conjC_v c1 c2)

 | DisjC_vC : forall e Q c1 c2,
    vcondtype e Q c1 →
     vcondtype e Q c2 →
      vcondtype e Q (disjC_v c1 c2)

 | ChcC_vC : forall e e' Q c1 c2,
    vcondtype (e ∧ (F) e') Q c1 →
     vcondtype (e ∧ (F) ( (F) e')) Q c2 →
      vcondtype e Q (chcC e' c1 c2).

Notation "{ e , Q |- vc }" := (vcondtype e Q vc) (e at level 200).
```

Note that, `InVR` in `Relation_vE` rule is the Coq encoding of *InVR* in Definition 3.2.5. InVR $r(A_v)^{e'}$ S_v states that there exists e such that $r(A_v)^e$ is in S_v, that is, $r(A_v)^e \in S_v$ and e' encodes both presence condition of relation r and feature model of v-schema S_v, that is, $e' = e \wedge pc(S_v)$.

Definition 3.2.5 (InVR). *InVR $r(A_v)^{e'}$ S_v states that $\exists e.r(A_v)^e \in S_v$ and $e' = e \wedge pc(S_v)$.*

Below is the Coq encoding of InVR, encoded as `InVR`.

```
(* InVR *)
Definition InVR (vr:vrelS) (vs:vschema) : Prop :=
let rn := getr vr in
 let vas := getvs vr in
  let e':= getf vr in
   exists e, In (rn, (vas, e)) (fst vs) ∧  (e ∧ (F) (snd vs)) = e'.
```

Consequently, in `Relation_vE` rule, *e'* in `InVR (rn, (A, e')) S` encodes both presence condition of relation `r` and feature model of v-schema S_v. Other typing rules in Coq encoding are straightforward encoding of typing rules in Figure 3.4.

The function used to compute the type of plain queries is denoted by $. \mid\mid= \ .$ and its definition and Coq encoding, `type_` is included in Appendix B.1. $S \mid\mid= \ q$ returns the type of the plain query q with underlying pain schema S in RA type system.

3.3 Correctness of VRA Type System

$$\begin{tikzcd} q_v \arrow[r, "type_v"] \arrow[d, dashed, "\mathbb{Q}[\![\cdot]\!]_c"'] & {A_v}^e \arrow[d, dashed, "\mathbb{QT}[\![\cdot]\!]_c"] \\ q \arrow[r, "type"] & A \end{tikzcd}$$

The VRA type system extends the RA type system to variational queries. In order to be correct with regard to the RA's type system, it must preserve variation encoded in a v-query. In other words, under same configuration, the configured v-type of a v-query in VRA type system should be equivalent to the plain type of the configured v-query in RA type system. For example, if in a variational context e with v-schema S_v, a v-query q_v has v-type ${A_v}^{e'}$, then for all configurations c, with plain schema $\mathbb{S}[\![S_v]\!]_c$, configured v-query, $\mathbb{Q}[\![q_v]\!]_c$ must have a plain type equivalent to $\mathbb{QT}[\![{A_v}^{e'}]\!]_c$. In the diagram on the right, $type_v$ refers to VRA type system, that is, $e, S_v \vDash q_v : {A_v}^{e'}$ and *type* refers to RA type system, that is, $S \mid\mid= \ q : A$ where $S = \mathbb{S}[\![S_v]\!]_c$. Also, vertical arrows represent corresponding configuration functions. Theorem 3.3.1 states the variation preservation property of VRA type system.

Theorem 3.3.1. *If a v-query q_v has v-type ${A_v}^{e'}$, then for all configurations c, $\mathbb{Q}[\![q_v]\!]_c$ has equivalent type to $\mathbb{QT}[\![{A_v}^{e'}]\!]_c$ i.e.,*

$$\forall c.\ \{e, S_v \vDash q_v : {A_v}^{e'}\} \quad \rightarrow \quad \mathbb{S}[\![S_v]\!]_c \mid\mid = \ \mathbb{Q}[\![q_v]\!]_c \ \equiv_{set} \ \mathbb{QT}[\![{A_v}^{e'}]\!]_c.$$

Theorem 3.3.1 is encoded in Coq as `variation_preservation` and its formal proof is included in the Appendix B.2.

```
Theorem variation_preservation : forall e S vq A' e',
   { e , S |= vq | (A', e') } →
   forall c, E[[e]]c = true →
      (S[[ S]]c) ||= (Q[[ vq]]c) =set= QT[[ (A', e')]]c.
Proof. (See Appndix  B.2 ). Qed.
```

Together with RA's type safety [30], variation preserving property of VRA type system implies that VRA type system is type safe as well.

One important thing to note here is that the variational conditions typing rules (Figure 3.5) are encoded as it is in [5]. However, the rules AttOptVal-C and AttOptAtt-C are not variation preserving over variation elimination with respect to respective plain condition typing rules. The problem is that these rules do not take variation associated with attributes in the variational schema of the VDB into account. One way they could be fixed to make them variation preserving is to require attribute references to be wrapped in choices that encode the variation information from the schema. This change is reflected in the following modified rules for variational conditions.

$$\textsc{AttOptVal-C}\quad \frac{a^{e'} \in A_v \qquad k \in dom_{\mathcal{I}}(a)}{e, A_v \vdash e'\langle a \bullet k,\ \texttt{false}\rangle} \qquad \textsc{AttOptAtt-C}\quad \frac{{a_1}^{e_1} \in A_v \qquad {a_2}^{e_2} \in A_v \qquad type(a_1) = type(a_2)}{e, A_v \vdash (e_1 \wedge e_2)\langle a_1 \bullet a_2,\ \texttt{false}\rangle}$$

However, this modification would make using v-queries more cumbersome since it requires many extra choices. Instead, the problem is solved by changing the corresponding plain condition typing rules (Appendix B.1) to make the Select-E rule in VRA variation preserving with respect to our modified version of RA.

Chapter 4: Formal Encoding of Implicitly Annotated Variational Queries

VDBMS provides an implicit way of writing v-queries to relieve the user from providing information that can be inferred automatically. When writing an implicitly annotated v-query, users do not need to include variation that is already encoded in the VDB and in the sub-queries and only required to include any further constraint they want to impose, if any. Recall that as stated in Section 3.2, presence conditions of any attribute, a in Q_v of v-query $\pi_{Q_v} q_v$ need to be at least as specific as a's presence condition in the type of q_v where type of q_v describes v-attributes present in its result. This requires user to repeat variation when writing v-queries. For example, in the example VDB used in Section 3.2, the v-schema is described by $S_{v3} = \{r\{{a_1}^{e_1}, {a_2}^{true}\}^{e_2}\}^{true}$ and $q_{v3} = \pi_{\{{a_1}^{e_1 \wedge e_2 \wedge e_3}\}^{\texttt{true}}}\ r$ is a valid variational projection query with respect to S_{v3}. Note that, q_{v3} includes a's presence condition $(e_1 \wedge e_2)$ in r. If implicit annotation is allowed, q_{v3} can be written as $\pi_{{a_1}^{e_3}} r$ omitting this information. Let's consider another example. Assume a VDB with v-schema $S_{v4} = \{r\{{a_1}^{e_1}, {a_2}^{e_2}\}^{e_3}\}^{true}$. $q_{v4} = \pi_{\{{a_1}^{true}\}^{true}}\ r$ and $q_{v5} = \pi_{\{{a_1}^{e_4}\}^{true}}\ r$ are valid implicitly annotated v-queries with respect to S_{v4}. The explicitly annotated versions of q_{v4} and q_{v5} are $\pi_{\{{a_1}^{e_1 \wedge e_2 \wedge e_3}\}^{true}}\ r$ and $\pi_{\{{a_1}^{e_1 \wedge e_2 \wedge e_3 \wedge e_4}\}^{true}}\ r$ respectively. $q_{v6} = \sigma_{e_c \langle a_1 = a_2, true \rangle}$ is another valid implicitly annotated v-query with respect to S_{v4}. The explicitly annotated version of q_{v6} is $\sigma_{e_1 \wedge e_2 \wedge e_3 \wedge e_c \langle a_1 = a_2, true \rangle}$.

The implicitly annotated variational query language has the same syntax as VRA as in Figure 3.1. However, to type check an implicitly annotated v-query, the type system needs to account for implicitness. Type system for implicitly annotated v-query, called *implicit VRA type system* is provided in Section 4.1. In VDMS, type checked implicitly annotated v-queries get explicitly annotated by the system with a function, called *explicitly annotating function* as described in Section 4.2. Finally, Section 4.3 provides correctness theorems and corresponding formal proofs of implicit VRA type system with respect to VRA type system.

4.1 VRA Type System for Implicitly Annotated V-Queries

VRA type system for implicitly annotated v-queries is similar to the VRA type system except for the fact that it can account for implicitness. The type generated by implicit VRA type system are explicitly annotated. Typing rules of implicit VRA system are shown in Figure 4.1 and corresponding formal encoding, `vtypeImp` is in Appendix C.1.

The rule Relation-E is same as before. It gets the type of relation r from v-schema S_v, hence the type is explicitly annotated, that is, variation encoded in the v-schema is present in the type.

The rule Project-E is however different from the that in VRA type system. The subset condition is replaced by a more lenient property, called *subsumption*, defined in the following Definition 4.1.1.

Definition 4.1.1 (V-set subsumption)**.** *The v-set* X_{v1} is subsumed by *the v-set* X_{v2}*, denoted by* $X_{v1} \prec X_{v2}$*, iff* $\forall x^{e_1} \in X_{v1}.\ sat(e_1) \rightarrow\ \exists e_2.\ x^{e_2} \in X_{v2}$ *and* $sat(e_1 \wedge e_2)$*, i.e., for any plain element* x *in* X_{v1} *with presence condition* e_1*, if* e_1 *is satisfiable then* x *must also in* X_{v2} *with such presence condition that its conjuction with* e_1 *is satisfiable. For example,* $\{2^{true}, 3^{true}\} \prec \{2^{true}, 3^{f_1}, 4^{true}\}$ *where* $sat(f_1)$*, however,* $\{2^{true}, 3^{f_1}\} \not\prec \{2^{true}, 3^{\sim f_1}\}$*.*

V-set subsumption is encoded in Coq as `subsump_velems` as follows.

```
(* Variational Set Subsumption *)
Definition subsump_velems (A A': velems) :Prop :=
forall x e, In (ae x e) A ∧  sat e → exists e', In (ae x e') A' ∧  sat(e ∧ (F) e').
```

V-set subsumption is extended to v-query type, that is, to annotated v-set in the following Definition 4.1.2. This is called *V-query type subsumption.*

Definition 4.1.2 (V-query type subsumption)**.** *The v-query type* ${X_{v1}}^{e_1}$ is subsumed by *the v-set* ${X_{v2}}^{e_2}$*, denoted by* ${X_{v1}}^{e_1} \prec {X_{v2}}^{e_2}$*, iff* $\forall x^{e_{x1}} \in X_{v1}.\ sat(e)_1 \wedge e_{x1} \rightarrow\ \exists e_2.\ x^{e_{x2}} \in X_{v2}$ *and* $sat(e_1 \wedge e_{x1} \wedge e_{x2} \wedge e_2)$*,*

V-query type subsumption is encoded in Coq as `subsump_vqtype` as listed below.

```
(* V-query Type Subsumption *)
Definition subsump_vqtype ( X X': vqtype) : Prop :=
let (A, ea) := X in
let (A', ea') := X' in
 forall x e, In (ae x e) A ∧  sat (e ∧ (F) ea) →
  exists e', In (ae x e') A' ∧  sat (e ∧ (F) ea ∧ (F) e' ∧ (F) ea').
```

Implicit V-Query Typing Rules:

EmptyRelation-E
$$e, S_v \vdash \varepsilon : \{\}^{\texttt{false}}$$

Relation-E
$$\frac{r(A_v)^{e_r} \in S_v \qquad e' = e_r \wedge pc(S_v) \qquad sat\big(e \wedge e'\big)}{e, S_v \vdash r : {A_v}^{(e' \wedge e)}}$$

Project-E
$$\frac{e, S_v \vdash q_v : {A_v}'^{e'} \qquad Q_v \prec {A_v}'^{e'}}{e, S_v \vdash \pi_{Q_v}\ q_v : Q_v \cap {A_v}'^{e'}}$$

Select-E
$$\frac{e, S_v \vdash q_v : {A_v}^{e'} \qquad e, {A_v}^{e'} \vdash \theta_v}{e, S_v \vdash \sigma_{\theta_v}\ q_v : {A_v}^{e'}}$$

Choice-E
$$\frac{e \wedge e', S_v \vdash q_{v1} : {A_{v1}}^{e_1} \qquad e \wedge \sim e', S_v \vdash q_{v2} : {A_{v2}}^{e_2}}{e, S_v \vdash e\langle q_{v1}, q_{v2}\rangle : {A_{v1}}^{e_1} \cup {A_{v2}}^{e_2}}$$

Product-E
$$\frac{e, S_v \vdash q_{v1} : {A_{v1}}^{e_1} \qquad e, S_v \vdash q_{v2} : {A_{v2}}^{e_2} \qquad {A_{v1}}^{e_1} \cap {A_{v2}}^{e_2} = \{\}}{e, S_v \vdash q_{v1} \times q_{v2} : {A_{v1}}^{e_1} \cup {A_{v2}}^{e_2}}$$

SetOp-E
$$\frac{e, S_v \vdash q_{v1} : {A_{v1}}^{e_1} \qquad e, S_v \vdash q_{v2} : {A_{v2}}^{e_2} \qquad {A_{v1}}^{e_1} \equiv_T {A_{v2}}^{e_2}}{e, S_v \vdash q_{v1} \times q_{v2} : {A_{v1}}^{e_1}}$$

Implicit V-Condition Typing Rules (b: boolean tag, a: plain attribute, k: constant value):

Boolean-C
$$e, A_v \vdash b$$

AttOptVal-C
$$\frac{a^{e'} \in A_v \qquad k \in dom_{\mathcal{I}}(a)}{e, A_v \vdash a \bullet k}$$

AttOptAtt-C
$$\frac{{a_1}^{e_1} \in A_v \qquad {a_2}^{e_2} \in A_v \qquad type(a_1) = type(a_2)}{e, A_v \vdash a_1 \bullet a_2}$$

Neg-C
$$\frac{e, A_v \vdash \theta_v}{e, A_v \vdash \sim \theta_v}$$

Conjunction-C
$$\frac{e, A_v \vdash \theta_{v1} \qquad e, A_v \vdash \theta_{v2}}{e, A_v \vdash \theta_{v1} \wedge \theta_{v2}}$$

Disjunction-C
$$\frac{e, A_v \vdash \theta_{v1} \qquad e, A_v \vdash \theta_{v2}}{e, A_v \vdash \theta_{v1} \vee \theta_{v2}}$$

Choice-C
$$\frac{e \wedge e', A_v \vdash \theta_{v1} \qquad e \wedge \sim e', A_v \vdash \theta_{v2}}{e, A_v \vdash e'\langle \theta_{v1}, \theta_{v2}\rangle}$$

Figure 4.1: Implicit Variational Relational Algebra(VRA) and Variational Condition(V-Condition) Typing Relation. The typing rule of a join query is the combination of rules Select-E and Product-E.

Now, in a variational context, e with v-schema S_v, assuming a v-query q_v has type ${A_v'}^{e'}$, the projection v-query $\pi_{Q_v}\ q_v$ has type $Q_v \cap {A_v'}^{e'}$, given that, $Q_v \prec {A_v'}^{e'}$. $Q_v \prec {A_v'}^{e'}$ ensures that plain attributes in Q_v will be in some variants of ${A_v'}^{e'}$. The type of sub-query q_v is explicitly annotated, therefore, taking intersection of to be projected attributes Q_v with q_v's type explicitly annotates the plain attributes in Q_v and also filters out any plain attributes from Q_v that are not present q_v's type. Note that, it doesn't check if Q_v is subset of q_v's type. However, it doesn't affect the typing, that is, typing of projection operation by Implicit VRA type system is still equivalent to its typing in VRA type system as by definition of annotated v-set intersection (Definition 2.3.34), $Q_v \cap {A_v'}^{e'}$ is subset of ${A_v'}^{e'}$. Hence, the type is correct.

However, the actual reason for subset check in VRA type system is to ensure that all plain attributes in Q_v are present in ${A_v'}^{e'}$ with such presence conditions that in all query variants of q_v, to be projected plain set has the subset relationship with the set it is projecting from. To have subset relationship in all query variants, presence conditions of plain attributes in Q_v need to be at least as specific as their presence conditions in ${A_v'}^{e'}$. The implicitly annotated v-queries goes through an additional step of *explicitly annotating* of v-queries right after the type check (Section 4.2) which replaces Q_v in $\pi_{Q_v}\ q_v$ with $Q_v \cap {A_v'}^{e'}$ that maintains the subset property.

The rest of the operations generate types directly from the sub-queries types which are explicitly annotated. Hence, they do not require any changes from VRA type system. In essence, implicit VRA type system is allows more flexibility on the user side that VDBMS is equipped to handle to produce valid v-queries. Correctness of implicit VRA type system and explicitly annotating function with respect to VRA type system (Section 4.3) guarantees that the process works as expected.

4.2 Explicitly Annotating V-Queries

Implicitly annotated v-queries makes VDBMS easier to use by relieving user from repeating variation information that is already encoded in the v-schema and sub-queries. However v-queries still need to be explicitly annotated by the system otherwise when decoupled from the v-schema, v-queries would lose variation information. Teh explicit annotation is done immediately after v-queries pass through the implicit VRA type system and before they are sent to SQL generator. The function that VDMS uses to explicitly

Explicitly Annotating V-Queries:

$\lfloor . \rfloor_{S_v} : \mathbf{Q} \rightarrow \mathbf{S_v} \rightarrow \mathbf{Q}$
$\lfloor r \rfloor_{S_v} = r$
$\lfloor \sigma_{\theta_v} q_v \rfloor_{S_v} = \sigma_{\theta_v} \lfloor q_v \rfloor_{S_v}$
$\lfloor \pi_{Q_v} q_v \rfloor_{S_v} = \pi_{Q_v \cap {A_v}'^{e'}} \lfloor q_v \rfloor_{S_v} \quad \mathit{where} \quad S_v \vdash \lfloor q_v \rfloor_{S_v} : {A_v}'^{e'}$
$\lfloor q_{v1} \times q_{v2} \rfloor_{S_v} = \lfloor q_{v1} \rfloor_{S_v} \times \lfloor q_{v2} \rfloor_{S_v}$
$\lfloor e\langle q_{v1}, q_{v2} \rangle \rfloor_{S_v} = e\langle \lfloor q_{v1} \rfloor_{S_v}, \lfloor q_{v2} \rfloor_{S_v} \rangle$
$\lfloor q_{v1} \circ q_{v2} \rfloor_{S_v} = \lfloor q_{v1} \rfloor_{S_v} \circ \lfloor q_{v2} \rfloor_{S_v}$
$\lfloor \varepsilon \rfloor_{S_v} = \varepsilon$

Figure 4.2: Explicitly Annotating Implicitly Annotated Variational Queries(V-Queries) w.r.t. Variational Schema(V-Schema). V-queries passed to this function are assumed to be well-typed.

annotate a v-query q_v with respect to v-schema S_v, denoted by $\lfloor q_v \rfloor_{S_v}$, is in Figure 4.2. The function returns relation queries as they are. Parameter Q_v of the projection v-query $\pi_{Q_v} q_v$ gets intersected with the type of its explicitly annotated sub-query $\lfloor q_v \rfloor_{S_v}$. $S_v \vdash \lfloor q_v \rfloor_{S_v} : {A_v}'^{e'}$ describes the type of $\lfloor q_v \rfloor_{S_v}$ in an empty variational context with v-schema S_v. In the resulted explicitly annotated projection v-query $\pi_{Q_v \cap {A_v}'^{e'}} \lfloor q_v \rfloor_{S_v}$, the parameter $Q_v \cap {A_v}'^{e'}$ is subset of and explicitly annotated with respect to $\lfloor q_v \rfloor_{S_v}$'s type. For other v-queries, sub-queries are explicitly annotated within the same structure.

The explicitly annotating function is encoded in Coq as `ImptoExp` as below.

```
(* Explicitly Annotating Implicitly Annotated V-Queries w.r.t. V-Schema. *)
Fixpoint ImptoExp (vq: vquery) (vs:vschema) : (vquery) :=
match vq with
| (empty_v)               ⇒ empty_v
| (rel_v (rn, (A_, e_'))) ⇒ let vr := (findVR rn vs) in
                              (rel_v (rn, (getvs vr, getf vr)))
| (proj_v Q  vq)          ⇒ let vq_s := (ImptoExp vq vs) in
                             let (A', e') := vtypeImpNOTC (litB true) vs vq_s in
                              let QinterQ' := vqtype_inter_vq Q (A', e') in
                                proj_v QinterQ' vq_s
| (chcQ e'   vq1 vq2)     ⇒ chcQ e' (ImptoExp vq1 vs) (ImptoExp vq2 vs)
| (prod_v    vq1 vq2)     ⇒ prod_v (ImptoExp vq1 vs) (ImptoExp vq2 vs)
| (setU_v op vq1 vq2)     ⇒ setU_v op (ImptoExp vq1 vs) (ImptoExp vq2 vs)
| (sel_v c   vq)          ⇒ sel_v c (ImptoExp vq vs)
```

```
end.

Notation "[ vq ] vs" := (ImptoExp vq vs) (at level 90, left associativity).
```

In the formal encoding, explicitly annotating function `ImptoExp` uses `vtypeImpNOTC` to get the type of v-queries. Implicit VRA typing rules are formally encoded as inductive propositions. Following lemma proves that `vtypeImpNOTC` provides the same type as the implicit VRA type system for any type-checked v-query.

```
Lemma vtypeImpNOTC_correct : forall e vs vq vt {HRn: NoDupRn (fst vs)},
       {e, vs |- vq | vt} → (vtypeImpNOTC e vs vq) = vt.
Proof.
intros. induction H;
try(simpl vtypeImpNOTC);
try(rewrite IHvtypeImp);
try(rewrite (IHvtypeImp1 HRn); rewrite (IHvtypeImp2 HRn));
try(reflexivity); try(assumption);try(assumption).
- apply InVR_findVR in H.
  rewrite H. unfold getvs. unfold getf.
  simpl. reflexivity. assumption.
Qed.
```

Correctness of explicitly annotating function with respect to implicit VRA type system is formalized in the next Section 4.3.

4.3 Correctness of Implicit VRA Type System

Implicit VRA type system's correctness is formalized with respect to the VRA type system that if an implicitly annotated v-query is valid, that is, has a type in implicit VRA type system then its explicitly annotated version is valid in VRA type system with an equivalent type. Theorem 4.3.1 mathematically states the above statement. Remember that, ($\vdash$) indicates implicit VRA type system and ($\vDash$) indicates VRA type system.

Theorem 4.3.1. *In a variation context e with v-schema S_v, for all v-queries q_v,*

$$\{e, S_v \vdash q_v : A\} \rightarrow \exists A',\ \{e, S_v \vDash \lfloor q_v \rfloor_{S_v} : A'\} \text{ and } A \equiv_{vqtype} A'$$

Theorem 4.3.1 can be proved with two lemmas. The first lemma (Lemma 4.3.2) proves correctness of explicitly annotating function with respect to implicit VRA type system. For any v-query q_v, if it has a type in implicit VRA type system, then its explicitly annotated version $\lfloor q_v \rfloor_{S_v}$ has an equivalent type in implicit VRA type system.

Lemma 4.3.2. *In a variation context e with v-schema S_v, for all v-queries q_v,*

$$\{e, S_v \vdash q_v : A\} \rightarrow \exists A',\ \{e, S_v \vdash \lfloor q_v \rfloor_{S_v} : A'\} \text{ and } A \equiv_{vqtype} A'$$

Lemma 4.3.2 is encoded as `ImpQ_ImpType_ExpQ_ImpType` in Coq and the respective formal proof is included in Appendix C.2.1

```
Lemma ImpQ_ImpType_ExpQ_ImpType e S q A:
      { e , S |-  q   | A }  →
      exists A', { e , S |- [q]S | A' } ∧  A =vqtype= A'.
Proof. (See Appndix  C.2.1 ). Qed.
```

The second lemma states that any explicitly annotated v-query's validity in implicit VRA type system implies its validity in VRA type system with an equivalent type.

Lemma 4.3.3. *In a variation context e with v-schema S_v, for all v-queries q_v,*

$$\{e, S_v \vdash \lfloor q_v \rfloor_{S_v} : A\} \rightarrow \exists A',\ \{e, S_v \vdash \lfloor q_v \rfloor_{S_v} : A'\} \text{ and } A \equiv_{vqtype} A'$$

Lemma 4.3.3 is encoded as `ExpQ_ImpType_ExpQ_ExpType` in Coq and the respective formal proof is included in Appendix C.2.2

```
Lemma ExpQ_ImpType_ExpQ_ExpType e S q A (HndpQ: NoDupElemvQ q):
      { e , S |-  [q]S   | A }  →
      exists A', { e , S |= [q]S | A' } ∧  A =vqtype= A'.
Proof. (See Appndix  C.2.2 ). Qed.
```

Implicit VRA type system correctness theorem (Theorem 4.3.1) is a corollary of these two lemmas (Lemma 4.3.2 and 4.3.3). Theorem 4.3.1 is encoded in Coq as `ImpQ_ImpType_ExpQ_ExpType` and its formal proof is given below.

```
Theorem ImpQ_ImpType_ExpQ_ExpType e S q A (HndpQ: NoDupElemvQ q):
   { e , S |-  q   | A }  →
   exists A', { e , S |= [q]S | A' } ∧  A =vqtype= A'.
Proof. intro HImp.
(*
   HImp : {e, S |- q | A}
   --------------------------------------
   exists A' : vqtype, {e, S |= [q] S | A'} ∧  A =vqtype= A'
*)

(* From Lemma 4.3.2,
   HImp:{ e , S |-  q   | A } →  HExpQ:{ e , S |- [q]S | A'' } ∧  A =vqtype= A''
*)
apply ImpQ_ImpType_ExpQ_ImpType in HImp as HExpQ.
destruct HExpQ as [A'' [HExpQ HEqiv'']] ].
```

```
(* From Lemma 4.3.3,
   HExpQ:{ e , S |- [q]S | A'' } → HExp:{ e , S |= [q]S | A' } ∧ A''=vqtype=A'
*)
apply ExpQ_ImpType_ExpQ_ExpType in HExpQ as HExp; try auto.
destruct HExp as [A' [HExp HEqiv'] ].

exists A'.
(*
   HExp : {e, S |= [q] S | A'}
   HEqiv' : A'' =vqtype= A'
   HEqiv'' : A =vqtype= A''
   --------------------------------------
   {e, S |= [q] S | A'} ∧ A =vqtype= A'
*)
split.
(* Goal: {e, S |= [q] S | A'} *)
apply HExp.
(* Goal: A =vqtype= A' *)
transitivity (A''); assumption.
Qed.
```

Theorem 4.3.1 along with variation preserving theorem for VRA type system (Theroem 3.3.1) implies that Implicit VRA type system is also variation preserving. Theorem 4.3.4 below states the variation preservation property for Implicit VRA type system.

Theorem 4.3.4. *If a v-query q_v has v-type A, then for all configurations c, $\mathbb{Q}[\![\lfloor q_v \rfloor_{S_v}]\!]_c$ has equivalent type to $\mathbb{QT}[\![A]\!]_c$, i.e. $\{e, S_v \vdash q_v : A\} \ \rightarrow \ \mathbb{S}[\![S_v]\!]_c \ || = \ \mathbb{Q}[\![\lfloor q_v \rfloor_{S_v}]\!]_c \ \equiv_{set} \ \mathbb{QT}[\![A]\!]_c$.*

Above theorem is encoded in Coq as `variation_preservation_Imp` and its formal proof applies Theorem 4.3.1 and 3.3.1 in its premise subsequently to get to the conclusion.

```
Theorem variation_preservation_Imp e S q A (HndpQ: NoDupElemvQ q):
   { e , S |- q | A } →
   forall c, E[[e]]c = true →
     ||= (Q[[ [q]S]]c) =set= QT[[ A]]c.

Proof. intros HImp c He.
(*
   HImp : {e, S |- q | A}
   --------------------------------------
   ||= (Q[[ [q] S]] c) =set= QT[[ A]] c
*)
(* From Theroem 4.3.1,
   HImp : {e, S |- q | A} → {e, S |= [q] S | A'} ∧ A =vqtype= A' *)
apply ImpQ_ImpType_ExpQ_ExpType in HImp; try auto.
```

```
destruct HImp as [A' [HExpQ HEquiv] ].
(* From Theorem 3.3.1,
   {e, S |= [q] S | A'} → HExpQ : ||= (Q[[ [q] S]] c) =set= QT[[ A']] c *)
destruct A' as (A', e').
eapply variation_preservation with (c:=c) in HExpQ;
try auto.
(* A =vqtype= A' → HEquiv : QT[[ A]] c =set= QT[[ A']] c *)
apply configVQtype_equiv with (c:=c) in HEquiv.
(*
   HExpQ : ||= (Q[[ [q] S]] c) =set= QT[[ A']] c
   HEquiv : QT[[ A]] c =set= QT[[ A']] c
   --------------------------------------
   ||= (Q[[ [q] S]] c) =set= QT[[ A]] c
*)
symmetry in HEquiv.
transitivity (QT[[ A']] c); auto.
Qed.
```

Chapter 5: Related Work

Variational databases were developed in previous work [6, 7, 5]. This thesis extends the formalizations provided in this prior work and encodes all of the formalization in the Coq proof assistant. Most significantly, this thesis provides new formalizations of variational set properties defined in Section 2.3.1.1 and mechanized proofs of correctness of variational set union and intersection operations, VRA type system, and the process of handling implicitly annotated v-queries by the VDBMS.

Managing database variation in time or space has been studied extensively. Schema evolution and data migration are two well supported temporal variations [27, 11, 4, 32, 29]. Data integration [14] is a form of variation in space. In the context of SPL, where variation can occur in both time and space, temporal and structural variation are addressed independently by researchers. Temporal variation in SPL is addressed by adapting work on database evolution [18]. Work on structural variation focus on generating specialized schema for each software variant in SPL [2, 22, 20]. Like VDBMS, Humblet *et al.* [20] uses annotations that connect software features to schema elements. Abo Zaid and De Troyer [2] also uses annotative approach, but works at a higher level than VDBMS. However, unlike VDBMS, none of these work can be generalized to handle arbitrary forms of variation not do they support interaction among variation and allow writing queries that can express both temporal and structural variation in information need.

In variational databases, variational schemas and variation tables (Sections 2.3.2 and 2.3.3) are based on existing work on variational sets [16, 36] which is a part of broader variational data structure research that strives to support computing with variation at runtime [28, 36]. Variational queries (Section 3.1) are supported through *choice calculus*. Choice calculus is a formal language to represent and transform variation in software and other structured documents [35].

An increasing number of formal system verification like the one presented in this thesis has been carried out in recent years and has appeared as published case studies in literature. Some notable work in formal verification are as follows. One of the successfully commercialized verification projects is the certified C compiler, CompCert [25, 26], for which Leory's work [25] received the POPL test of the time award in 2016. CakeML, a variant of standard ML language [24] also comes with formally verified compiler. Klein *et al.* [23] provide correctness proofs for a seL4 Operating System Kernel. Gu *et al.* [17] formally certified a concurrent kernel for the x86 architecture that has fine-grained locking. Coq correctness proofs of an Raft distributed consensus protocol are presented in [31]. Blanchette *et al.* [9] provides formalization of conflict-driven clause learning calculus for sat solver. Amazon web service (AWS) is increasingly investing in formal verification to raise security level of its products [13] and has already formalized open source implementation of Transport Layer Service (TLS) Protocol that is used in numerous Amazon services [12].

Chapter 6: Conclusion and Future Work

Variation in data is unavoidable and can appear in many forms within different contexts. Consequently, variation management has been extensively studied by the database community including schema evolution, data integration, database versioning. However, while there are many efficient context specific solutions, no fundamental solution exists that can handle variation of any form, irrespective of the context. Moreover, in practice, variation of different forms can interact in a particular context. For example, temporal and spatial variation in database collide in the context of SPL with no good solution to support the interaction. Variational databases treat variation orthogonal to data and extend relational databases with explicit encoding of variation within the database through annotation. The elementary structure in the variation database is variational set. In the effort of formalizing variational databases, this thesis first formally encodes the variational set and its operations, formally defines and encodes variational set properties, and provides formal proofs of correctness of variational set operations. Correctness of the variational set operations is defined with respect to respective plain set operations. Then, it formally encodes variational database schema and content. In addition to incorporating variation into the database, VDBMS also provides variational queries, that is v-queries, that can explicitly express variation in its information need and allows writing implicitly annotated v-queries without repeating variation that are already encoded in the VDB and in the sub-queries. The variational relational algebra (VRA) of VDBMS for writing v-queries comes with a static type system. This thesis provides formally verification of the VRA type system as well as formally verifies the process of handling implicitly annotated v-query by the VDBMS.

Formal proofs of most of the theorems are included in the Appendix. Two proofs in Appendix C.2.6 and C.2.5 as well as the two, in Appendix C.2.4 and C.2.3 can be combined into one proof. Also, the variational condition typing rules (Figure 3.5) are encoded as they are defined in [5] where two typing rules are not variation preserving with respect to the standard plain condition typing rules (see the discussion in Section 3.2). Immediate follow-up of this work could be studying the consequence of making these typing rules variation preserving on the usability of v-query and modifying the formalization as such.

Bibliography

[1] *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] Lamia Abo Zaid and Olga De Troyer. Towards modeling data variability in software product lines. In Terry Halpin, Selmin Nurcan, John Krogstie, Pnina Soffer, Erik Proper, Rainer Schmidt, and Ilia Bider, editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 453–467, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines.* Springer-Verlag, Berlin, 2016.

[4] Gad Ariav. Temporally oriented data definitions: Managing schema evolution in temporally oriented databases. *Data & Knowledge Engineering*, 6(6):451 – 467, 1991.

[5] Parisa Ataei, Qiaoran Li, Eric Walkingshaw, and Arash Termehchy. Managing variability in relational databases by VDBMS. *CoRR*, abs/1911.11184, 2019.

[6] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. Variational databases. In *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*, pages 11:1–11:4, 2017.

[7] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. Managing structurally heterogeneous databases in software product lines. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB 2018 Workshops, Poly and DMAH, Rio de Janeiro, Brazil, August 31, 2018, Revised Selected Papers*, pages 68–77, 2018.

[8] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow.*, 8(12):1346–1357, August 2015.

[9] Jasmin Christian Blanchette, Mathias Fleury, and Christoph Weidenbach. A verified sat solver framework with learn, forget, restart, and incrementality. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 4786–4790, 2017.

[10] Goetz Botterweck and Andreas Pleuss. *Evolution of Software Product Lines*, pages 265–295. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[11] Cristina De Castro, Fabio Grandi, and Maria Rita Scalas. Schema versioning for multitemporal relational databases††recommended by peri loucopoulos. *Information Systems*, 22(5):249 – 290, 1997.

[12] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. Continuous formal verification of amazon s2n. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 430–446, Cham, 2018. Springer International Publishing.

[13] Byron Cook. Formal reasoning about the security of amazon web services. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 38–47, Cham, 2018. Springer International Publishing.

[14] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[15] Martin Erwig and Eric Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011.

[16] Martin Erwig, Eric Walkingshaw, and Sheng Chen. An Abstract Representation of Variational Graphs. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 25–32, 2013.

[17] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association.

[18] Kai Herrmann, Jan Reimann, Hannes Voigt, Birgit Demuth, Stefan Fromm, Robert Stelzmann, and Wolfgang Lehner. Database evolution for software product lines. In *DATA*, 2015.

[19] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya Parameswaran. Orpheusdb: Bolt-on versioning for relational databases. *Proc. VLDB Endow.*, 10(10):1130–1141, June 2017.

[20] Mathieu Humblet, Dang Vinh Tran, Jens H. Weber, and Anthony Cleve. Variability management in database applications. In *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*, VACE '16, pages 21–27, New York, NY, USA, 2016. ACM.

[21] Christian S. Jensen and Richard T. Snodgrass. *Temporal Query Languages*, pages 3009–3012. Springer US, Boston, MA, 2009.

[22] Niloofar Khedri and Ramtin Khosravi. Handling database schema variability in software product lines. In *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 1*, pages 331–338, 2013.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, and et al. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[24] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. *SIGPLAN Not.*, 49(1):179–191, January 2014.

[25] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):42–54, January 2006.

[26] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[27] E. McKenzie and Richard Thomas Snodgrass. Schema evolution and the relational algebra. *Inf. Syst.*, 15(2):207–232, May 1990.

[28] Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Int. Work. on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 28–35, 2017.

[29] Hyun J. Moon, Carlo A. Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and querying transaction-time databases under schema evolution. *Proc. VLDB Endow.*, 1(1):882–895, August 2008.

[30] Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, page 174–183, New York, NY, USA, 1988. Association for Computing Machinery.

[31] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[32] Richard Thomas Snodgrass. *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, USA, 1995.

[33] The Coq Development Team. The coq proof assistant, July 2020.

[34] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. Towards efficient analysis of variation in time and space. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, SPLC '19, page 57–64, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Eric Walkingshaw. The Choice Calculus: A Formal Language of Variation. In *PhD Dissertation.* Oregon State University, 2013. `http://hdl.handle.net/1957/40652`.

[36] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226, 2014.

APPENDICES

Appendix A: Formal Encoding of Variational Set

Variation sets are discussed in Section 2.3.1. Formal encoding VDB requires encoding of both plain set and v-set. Following sections include encoded definitions and mechanized proofs required for formalizing variational sets.

A.1 Plain Set and V-set Properties

A.1.1 nodupelem Equivalence Property

`nodupelem` converts any variational set to an equivalent variational set with *No-Dup-Elem* property. Following lemma `nodupelem_gen_equiv_velems_list` formally proves that this function maintains equivalency.

```
Lemma nodupelem_gen_equiv_velems_list: forall v, v =vlist= (nodupelem v).
Proof. intro v.
functional induction (nodupelem v) using nodupelem_ind;
unfold "=vlist="; unfold "=vlist=" in *; intro c; simpl.
+ reflexivity.
+ destruct (E[[ e]] c) eqn:He;
[apply cons_equiv_list | ]; auto.
+ destruct (E[[ e]] c) eqn:He.
(* (E[[ e]] c) = true *)
++ rewrite orb_true_l.
specialize IHv0 with c.
apply cons_equiv_list with (a:=a) in IHv0.
rewrite ← IHv0. rewrite existsbElem_InElem in e1.
unfold "=list=".
(* Goal: In a0 (a :: X[[ vs]] c) ↔ In a0 (a :: X[[ removeElem a vs]] c) *)
intro a0. split; intro.
+++ (* → *) simpl in H. destruct H. rewrite H. simpl. eauto.
destruct (string_dec a0 a) eqn:Haa0.
rewrite e0. simpl. eauto.
apply removeElem_neq_In with (vs:=vs) (c:=c) in n as HInrm.
simpl. rewrite HInrm in H. eauto.
+++ (* ← *) simpl in H. destruct H. rewrite H. simpl. eauto.
destruct (string_dec a0 a) eqn:Haa0.
rewrite e0. simpl. eauto.
apply removeElem_neq_In with (vs:=vs) (c:=c) in n as HInrm.
```

```
simpl. rewrite ← HInrm in H. eauto.
(* (E[[ e]] c) = false *)
++ simpl. destruct (E[[ get_annot a vs]] c) eqn:Hget.
+++ (* (E[[ get_annot a vs]] c) = false *) specialize IHv0 with c.
apply cons_equiv_list with (a:=a) in IHv0.
rewrite ← IHv0. apply get_annot_true_In in Hget.
apply removeElem_In in Hget. auto.
+++ (* ((E[[ get_annot a vs]] c) = false *) rewrite ← IHv0.
apply get_annot_false_notIn in Hget.
apply removeElem_notIn in Hget. auto.
Qed.
```

A.1.2 Plain Set Equivalence Relation

Plain set equivalence in Defintion 2.3.11 is encoded as `equiv_elems` in Coq (See Section 2.3.1.1). Following are the Coq proofs of its equivalence relation property.

```
(* equiv_elems is Reflexive *)
Remark equiv_elems_refl: Reflexive equiv_elems.
Proof.
intros A a. split; reflexivity.
Qed.

(* equiv_elems is Symmetric *)
Remark equiv_elems_sym : Symmetric equiv_elems.
Proof.
intros A A' H a.
split; symmetry;
apply H.
Qed.

(* equiv_elems is Transitive *)
Remark equiv_elems_trans : Transitive equiv_elems.
Proof.
intros A A'' A' H1 H2 a.
split; try (transitivity (In a A''));
try (transitivity (count_occ string_eq_dec A'' a));
try (apply H1);
try (apply H2).
Qed.

(* equiv_elems is an Equivalence relation *)
Instance elems_Equivalence : Equivalence equiv_elems := {
Equivalence_Reflexive := equiv_elems_refl;
Equivalence_Symmetric := equiv_elems_sym;
Equivalence_Transitive := equiv_elems_trans }.
```

A.1.3 V-set Equivalence Relation

V-set equivalence in Defintion 2.3.12 is encoded as `equiv_velems` in Coq (See Section 2.3.1.1). Following are the Coq proofs of its equivalence relation property.

```
(* equiv_velems is Reflexive *)
Remark equiv_velems_refl: Reflexive equiv_velems.
Proof.
intros A a. reflexivity.
Qed.

(* equiv_velems is Symmetric *)
Remark equiv_velems_sym : Symmetric equiv_velems.
Proof.
intros A A' H a.
symmetry.
apply H.
Qed.

(* equiv_velems is Transitive *)
Remark equiv_velems_trans : Transitive equiv_velems.
Proof.
intros A A'' A' H1 H2 a.
transitivity (configVElemSet A'' a).
apply H1.
apply H2.
Qed.

(* equiv_velems is a Equivalence relation *)
Instance velems_Equivalence : Equivalence equiv_velems := {
Equivalence_Reflexive := equiv_velems_refl;
Equivalence_Symmetric := equiv_velems_sym;
Equivalence_Transitive := equiv_velems_trans }.
```

A.1.4 Annotated V-set Equivalence Relation

Annotated V-set equivalence in Defintion 2.3.13 is encoded as `equiv_avelems` in Coq (See Section 2.3.1.1). Following are the Coq proofs of its equivalence relation property.

```
(* equiv_avelems is Reflexive *)
Remark equiv_avelems_refl: Reflexive equiv_avelems.
```

```
Proof.
intro X. destruct X. unfold equiv_avelems. split;
reflexivity.
Qed.

(* equiv_avelems is Symmetric *)
Remark equiv_avelems_sym : Symmetric equiv_avelems.
Proof.
intros X Y. intros H. destruct X, Y. unfold equiv_avelems.
unfold equiv_avelems in H. symmetry. apply H.
Qed.

(* equiv_avelems is Transitive *)
Remark equiv_avelems_trans : Transitive equiv_avelems.
Proof.
intros X Y Z. intros H1 H2.
destruct X as (vx, fx), Y as (vy, fy), Z as (vz, fz).
unfold equiv_vqtype in H1.
unfold equiv_vqtype in H2.
unfold equiv_vqtype.
intro c. transitivity (QT[[ (vy, fy)]] c); auto.
Qed.

(* equiv_avelems is a Equivalence relation *)
Instance avelems_Equivalence : Equivalence equiv_avelems := {
Equivalence_Reflexive := equiv_avelems_refl;
Equivalence_Symmetric := equiv_avelems_sym;
Equivalence_Transitive := equiv_avelems_trans }.
```

A.1.5 V-Set Subset Correctness

```
Theorem subset_velems_correctness A A' (HndpA: NoDupElem A) (HndpA': NoDupElem
A'):
        subset_velems_exp A A' ↔ (forall c, subset (X[[ A]]c) (X[[ A']]c)).
 Proof. split;
 generalize dependent A'; generalize dependent A;
 induction A' as [|(a', ea') A' IHA'];
 intros HndpA' H.

 (*Goals → :  1: A' := [] , 2: A' := [ae a ' ea':A']*)
 1, 2: unfold subset_velems_exp in H; unfold subset; intros c x;

 try (split; (* 1: subset A []      to 1-1: In A []        1-2: count A [] *)
             (* 2: subset A [_:A'] to 2-1: In A [_:A'][] 2-2: count A [_:A'][] *)
```

```
[ (* 1-1 2-1 In: intro In x X[[A]]c  *)
  intro HInxA |

  (* 1-2 2-2 count: destruct (count_occ A x) *)
  destruct (count_occ string_eq_dec (X[[ A]] c) x) eqn:Hcount;
  [(* Case 0: count_occ string_eq_dec (X[[ A]] c) x = 0 *)
   (* trivial 0 <= any *) simpl; auto; apply (count_occ_ge_0) |

   (* Case Sn: count_occ l x = S n → HInxA: In x X[[A]]c *)
   pose (gt_Sn_O n) as HInxA; rewrite ← Hcount in HInxA;
   rewrite ← count_occ_In in HInxA ]
]); (* 1-1 →1 , 2-1 → 2, 2-1 → 3, 2-2 → 4*)

(* In x X[[A]]c → exists e, In (x, e) A ∧ E[[e]]c = true *)
rewrite ← In_config_exists_true in HInxA;
destruct HInxA as [e HInxeA];
specialize H with x e c;
(* cereate subset premise H': In (x,e) A ∧ sat e*)
assert (H': In (ae x e) A ∧ (E[[ e]] c) = true);
try(auto);
(* get subset conclusion with H' → HIne': In (x, e') A' ∧ Hsat: e → e' *)
apply H in H'; destruct H' as [e' He'];
destruct He' as [HIne' Hsat]; simpl in HIne'.

(* 1, 2: In x [] , count x []
         destruct (In (x, e') []) *)
1, 2: try (destruct HIne'). (* proving 1,2 changes goals* 3→1, 4→2 *)

(* 1, 2: In x [_:A'] , count [_:A'] x *)
1, 2: destruct HInxeA as [HInxeA Hetrue];
(* (E[[ e]] c) = true → (E[[ e']] c) = true*)
(*rewrite not_sat_not_prop in Hsat;
rewrite ← sat_taut_comp in Hsat;
specialize  Hsat with c; apply Hsat in Hetrue;*)

(* 1: In x (X[[ ae a' ea' :: A']] c) →  (x = a' ∧ ea' = true) ∨ In x X[[A']]c *)
(* 2: count_occ (X[[ ae a' ea' :: A']] c) x →
                    [case (x = a' ∧ ea' = true): S (count_occ X[[A']]c x)
                     case  _                   :    count_occ X[[A']]c x *)

(* destruct HIne': (ae a' ea' = ae x e' ∨ In (ae x e') A') *)
try (destruct HIne' as [Heq | Hin];

[(* Case Heq: ae a' ea' = ae x e' *)
 inversion Heq; subst;
 simpl; rewrite Hsat; simpl
 |
 (* Case HIn: In (ae x e') A'*) (* proves Goal 1 right *)

]). (** 1→  1(Heq), 2(HIn)  2→  3(Heq), 4(HIn)*)

3, 4: apply NoDupElem_NoDup_config with (c:=c) in HndpA as Hcount';
rewrite (NoDup_count_occ string_eq_dec) in Hcount'; specialize Hcount' with x;
assert (Hn: n = 0); try (Lia.lia); rewrite Hn;

inversion HndpA'; subst; apply notInElem_notIn_config with (c:=c) in H2;
rewrite (count_occ_not_In string_eq_dec) in H2.

{ (* 1: 1- Case Heq *)left. reflexivity. }
{ (* 2: 1- Case HIn *)simpl. destruct (E[[ ea']] c); try simpl;
                     try right; rewrite ← In_config_exists_true; exists e';
                     eauto. }

{ (* 3: 2- Case Heq *)case (string_eq_dec x x); intro Hx; [|contradiction].
                     Lia.lia. }
{ (* 4: 2- Case HIn *)simpl. destruct (E[[ ea']] c); try simpl;
                     [ case (string_eq_dec a' x); intro; [ Lia.lia | ] |];

                      assert (HInxA': In x (X[[ A']]c));
                      try(rewrite ← In_config_exists_true; exists e'; eauto);
                      rewrite (count_occ_In string_eq_dec) in HInxA';
                      apply NoDupElem_NoDup_config with (c:=c) in H4 as HcountA';
                      rewrite (NoDup_count_occ string_eq_dec) in HcountA';
                      specialize HcountA' with x; Lia.lia. }

(* ← *)

(* case []: *)

(** Prove with two facts: subset (X[[A]]c) [] → subset_velems_exp A []
1. forall c, (X[[A]]c)  = [] → subset_velems_exp A []
2. exists c, (X[[A]]c) <> [] →   subset (X[[A]]c) [] *)

(* introduce (forall c, (X[[A]]c) = []) ∨ (exists c, (X[[A]]c) <> []) *)
pose Classical_Prop.classic as Hclassic.
specialize Hclassic with (forall c, (X[[ A]] c) = []).

destruct Hclassic as [Hall | Hexists].

{ (* case 1: forall c, (X[[A]]c) = [] *)
  apply nilconfig_subset_nil. assumption. }

{ (* case 2: exists c, (X[[A]]c) <> [] *)
  apply not_all_ex_not in Hexists. destruct Hexists as [c Hexists].
  specialize H with c.
  destruct ((X[[ A]] c)) eqn: HAc. contradiction. simpl in H.
  apply not_subset_cons_nil in H. destruct H. }

(* case (ae a' ea': A'):  *)
unfold subset_velems_exp. intros x e c HInxeA.
destruct HInxeA as[ HInxeA Hsat].
unfold subset in H.

specialize H with c x.
destruct H as [HInxAA' Hcount].
assert (HInxA: In x (X[[A]]c)).
rewrite ← In_config_exists_true. exists e. eauto.
apply HInxAA' in HInxA as HInxA'. simpl in HInxA'.
destruct (E[[ ea']] c) eqn: Hea;

[ simpl in HInxA';
  destruct HInxA' as [Heq | HInxA'];
  [ exists ea';  inversion Heq; subst; split;
    [simpl; left; reflexivity | auto] | ] | ].

1, 2: rewrite ← In_config_exists_true in HInxA';
destruct HInxA' as [e' HInxA'];
destruct HInxA' as [HInxe'A' He'];
exists e'; split;
[simpl; right; auto | auto].

Qed.
```

A.2 Plain Set and V-set Operations

V-set operations are defined to extend plain set operations for variational sets. Following sections provide formal proofs of expected properties of these operation.

A.2.1 Plain Set Union Identity

```
(* Plain set union nil-r *)
Lemma elems_union_nil_r: forall A, elems_union A [] =set= A.
Proof. intros. simpl. reflexivity. Qed.

(* Plain set union nil-l *)
Lemma elems_union_nil_l: forall A (H: NoDup A), elems_union [] A =set= A.
Proof. intros. unfold elems_union. unfold equiv_elems.
```

```
intro a. split. split.
- rewrite set_union_iff. simpl. intro.
destruct H0. destruct H0. auto.
- intro H0. rewrite set_union_iff. eauto.
- pose (NoDup_nil elem) as Hnil.
pose (set_union_nodup string_eq_dec Hnil H) as Hndp.
destruct (in_dec string_eq_dec a A).
+ apply (set_union_intro2 string_eq_dec) with (x:=[]) in i as HsetU.
rewrite NoDup_count_occ' in Hndp, H.
rewrite Hndp, H.
reflexivity. exact i. exact HsetU.
+ assert (n':  In a []). simpl. unfold not. eauto.
pose (conj n' n) as Hnn'.
rewrite notIn_set_union in Hnn'.
rewrite count_occ_not_In in n, Hnn'.
rewrite n, Hnn'. reflexivity.
Qed.
```

A.2.2 V-Set Union Identity

```
(* V-set union nil-r *)
Lemma velems_union_nil_r : forall A (H: NoDupElem A), velems_union A [] = A.
Proof. intro A.
intro H. unfold velems_union. simpl.
apply nodupelem_fixed_point. auto.
Qed.

(* V-set union nil-l *)
Lemma velems_union_nil_l : forall A (H: NoDupElem A), velems_union [] A =vset= A.
Proof.
induction A. simpl.
reflexivity. simpl.
intro H.
unfold velems_union.
simpl. destruct a as (a, e).
(* get   In a set_union from context and rewrite set_add *)
pose (NoDupElem_NoDup H) as Hnodup.
inversion Hnodup; subst.
apply (contrapositive _ _ (set_union_emptyL velem_eq_dec (ae a e) A)) in H2.
unfold set_In in H2.
pose (notIn_set_add_equiv_velems (ae a e) (set_union velem_eq_dec [] A)) as
Hset_add.
rewrite (nodupelem_equiv (Hset_add (H2))).
clear Hnodup.
(* get   InElem a set_union from context and rewrite nodupelem *)
```

```
inversion H; subst.
assert(HInnil:  InElem a []). simpl. eauto.
pose (conj HInnil H4) as H4'.
apply (notInElem_set_union a [] A) in H4'.
pose nodupelem_not_in_cons as Hcons.
apply (Hcons a e ((set_union velem_eq_dec [] A))) in H4'.
rewrite H4'.
apply IHA in H6.
unfold velems_union in H6.
apply cons_equiv_velems. assumption.
Qed.
```

A.2.3 V-set Union Variation Preservation

V-set union variation preservation theorem is given below. A mathematical proof is included for this theorem as its formal proof is a bit complicated than others and also to demonstrate the correspondence between mathematical and formal proofs at least once in the thesis. The mathematical proof is followed by the respective formal proof.

Theorem A.2.1. *For any two* v-sets, X_v *and* $X_v{}'$, $\mathbb{X}[\![X_v \cup X_v{}']\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v{}']\!]_c$.

Proof. Proof by induction on $X_v{}'$.

- Base case $X_v{}' = \{\}$: Proof by cases of X_v.
  - case $X_v = \{\}$: $\mathbb{X}[\![\{\} \cup \{\}]\!]_c \equiv_{set} \mathbb{X}[\![\{\}]\!]_c \cup \mathbb{X}[\![\{\}]\!]_c \Rightarrow \{\} \equiv_{set} \{\} \Rightarrow True$ (*reflexivity*)
  - case $X_v = (a^e :: X_v)$:

$$\begin{array}{llr} & \mathbb{X}[\![(a^e :: X_v) \cup \{\}]\!]_c \equiv_{set} \mathbb{X}[\![(a^e :: X_v)]\!]_c \cup \mathbb{X}[\![\{\}]\!]_c & \\ => & \mathbb{X}[\![\{\}]\!]_c \equiv_{set} \mathbb{X}[\![\{\}]\!]_c & (From\ lemma\ 2.3.21\ and\ 2.3.23) \\ => & True & (reflexivity) \end{array}$$

- Inductive case $X_v{}' = (a'^{e'} :: X_v{}')$:
  From inductive hypothesis,
  $$\forall X_v,\ \mathbb{X}[\![X_v \cup X_v{}']\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v{}']\!]_c \tag{2.1}$$
  we need to prove t,hat i.e. our goal is,
  $$\mathbb{X}[\![X_v \cup (a'^{e'} :: X_v{}')]\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![(a'^{e'} :: X_v{}')]\!]_c \tag{2.2}$$

  From *No-Dup-Elem* property (Definition 2.3.1) of v-set $(a'^{e'} :: X_v{}')$, we get,
  $$\sim \textit{In-Elem}\ \ a'\ X_v{}' \tag{2.3}$$
  $$\sim \textit{In}\ \ a'^{e'}\ X_v{}' \tag{2.4}$$

  Proof by cases ($In\ \ a'^{e'}\ X_v$).

– case$In \;\; a'^{e'} \;X_v$:

$$In \;\; a'^{e'} \; X_v \tag{2.5}$$

From eq. 2.5 we get,

$$\mathbb{X}[\![X_v \cup (a'^{e'} :: X_v')]\!]_c \equiv_{set} \mathbb{X}[\![X_v \cup X_v']\!]_c \tag{2.6}$$

From eq. 2.6, our goal in eq. 2.2 becomes,

$$\begin{aligned} & \mathbb{X}[\![X_v \cup X_v']\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![(a'^{e'} :: X_v')]\!]_c \\ => \;\; & \mathbb{X}[\![X_v \cup X_v']\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup (if \; \mathbb{E}[\![e']\!]_c \; then \; (a :: \mathbb{X}[\![X_v']\!]_c) \; else \; \mathbb{X}[\![X_v']\!]_c) \quad (From \; Defintion \; (Fig. \; 2.4)) \end{aligned} \tag{2.7}$$

Proof by cases of $\mathbb{E}[\![e']\!]_c = \mathit{True}$.

∗ case $\mathbb{E}[\![e']\!]_c = \mathit{True}$:
Eq. 2.7 becomes,

$$\mathbb{X}[\![X_v \cup X_v']\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup (a' :: \mathbb{X}[\![X_v']\!]_c) \tag{2.8}$$

Simplifying plain set union on the R.S. gives,

$$\mathbb{X}[\![X_v \cup X_v']\!]_c \equiv_{set} \textit{Set-add } a' \; (\mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c) \tag{2.9}$$

From Eq. 2.5, we get,

$$\begin{aligned} & In \; a'^{e'} \; X_v \\ => \quad & In \; a' \; \mathbb{X}[\![X_v]\!]_c \\ => \quad & In \; a' \; (\mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c) \\ => \quad & Set \; add \; a' \; (\mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c) = \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c \end{aligned} \tag{2.10}$$

Rewriting equation 2.10 in eq. 2.9,

$$\mathbb{X}[\![X_v \cup X_v']\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c \tag{2.11}$$

which is $True$ by induction hypothesis (Eq. 2.1).

∗ case $\mathbb{E}[\![e']\!]_c = \mathit{False}$:
Eq. 2.7 becomes,

$$\mathbb{X}[\![X_v \cup X_v']\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup (a' :: \mathbb{X}[\![X_v']\!]_c) \tag{2.12}$$

which is $True$ by induction hypothesis (Eq. 2.1).

– case $\sim In \;\; a'^{e'} \; X_v$:

$$\sim In \;\; a'^{e'} \; X_v \tag{2.13}$$

Even if variational element $a'^{e'}$ not in X_v, plain element a' can still be in X_v with some other annotation or can be absent.
Proof by cases of (*In-Elem* a' X_v).

∗ case *In-Elem* a' X_v:

$$\textit{In-Elem} \;\; a' \; X_v \tag{2.14}$$

From eq. 2.13 and 2.14,

$$X_v \cup (a'^{e'} :: X_v') \equiv_{vset} a'^{\; e' \; \vee \; \textit{get-annot } a' \; (X_v \cup X_v')} :: ((\textit{remove-elem } a' \; X_v) \cup X_v') \tag{2.15}$$

L.S. of our goal (Eq. 2.2),

$$\begin{aligned} L.S. \quad &\equiv_{set} \mathbb{X}[\![X_v \cup ({a'}^{e'} :: {X_v}')]\!]_c \\ &\equiv_{set} \mathbb{X}[\![{a'}^{\,e' \,\vee\, \textit{get-annot}\ a'\ (X_v \cup {X_v}')} :: ((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \quad (\textit{Rewriting eq.}2.15) \\ &\equiv_{set} \textit{if}\ (\mathbb{E}[\![\ e' \ \vee\ \textit{get-annot}\ a'\ (X_v \cup {X_v}')]\!]_c) \\ &\qquad \textit{then}\ a' :: \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \\ &\qquad \textit{else}\ \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \qquad (\textit{From Defintion (Fig.}\ 2.4)) \end{aligned} \tag{2.16}$$

R.S. of our goal (Eq. 2.2),

$$\begin{aligned} R.S. \quad &\equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![({a'}^{e'} :: {X_v}')]\!]_c \\ &\equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup (\textit{if}\ \mathbb{E}[\![e']\!]_c\ \textit{then}\ (a' :: \mathbb{X}[\![{X_v}']\!]_c)\ \textit{else}\ \mathbb{X}[\![{X_v}']\!]_c) \quad (\textit{From Definition (Fig.}\ 2.4)) \end{aligned} \tag{2.17}$$

Proof by cases of ($\mathbb{E}[\![e']\!]_c$).

- case $\mathbb{E}[\![e']\!]_c = \textit{True}$:

  From eq. 2.16 and 2.17 we get,

$$\begin{aligned} L.S. \qquad &\equiv_{set} \textit{if}\ (\mathbb{E}[\![\textit{True} \vee\ \textit{get-annot}\ a'\ (X_v \cup {X_v}')]\!]_c) \\ &\qquad \textit{then}\ a' :: \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \\ &\qquad \textit{else}\ \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \\ &\equiv_{set} a' :: \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \end{aligned} \tag{2.18}$$

$$R.S. \qquad \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup (a' :: \mathbb{X}[\![{X_v}']\!]_c) \tag{2.19}$$

  Hence, our current goal is,

$$a' :: \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup (a' :: \mathbb{X}[\![{X_v}']\!]_c) \tag{2.20}$$

  From Eq. 2.3 we get,

$$\mathbb{X}[\![X_v]\!]_c \cup (a' :: \mathbb{X}[\![{X_v}']\!]_c) \equiv_{set} a' :: (\mathbb{X}[\![\textit{remove-elem}\ a'\ X_v]\!]_c \cup \mathbb{X}[\![{X_v}']\!]_c) \tag{2.21}$$

  Applying transitivity on eq. 2.20 and 2.21, we get,

$$a' :: \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \equiv_{set} a' :: (\mathbb{X}[\![\textit{remove-elem}\ a'\ X_v]\!]_c \cup \mathbb{X}[\![{X_v}']\!]_c) \tag{2.22}$$

  which is $True$ by induction hypothesis (Eq. 2.1).

- case $\mathbb{E}[\![e']\!]_c = \textit{False}$:

  From eq. 2.16 and 2.17 we get,

$$\begin{aligned} L.S. \qquad &\equiv_{set} \textit{if}\ (\mathbb{E}[\![\ \textit{get-annot}\ a'\ (X_v \cup {X_v}')]\!]_c) \\ &\qquad \textit{then}\ a' :: \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \\ &\qquad \textit{else}\ \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \end{aligned} \tag{2.23}$$

$$R.S. \qquad \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![{X_v}']\!]_c \tag{2.24}$$

  If,

$$\mathbb{E}[\![\ \textit{get-annot}\ a'\ (X_v \cup {X_v}')]\!]_c = \textit{True} \tag{2.25}$$

  from eq. 2.23 and 2.24 we get,

$$a' :: \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup {X_v}')]\!]_c \equiv_{set}\ \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![{X_v}']\!]_c \tag{2.26}$$

From definition of *get-annot* (Def. 2.3.4),

$$\begin{aligned}\mathbb{E}[\![\ \textit{get-annot}\ a'\ (X_v \cup X_v')]\!]_c = \textit{True} &=> \textit{In-Elem}\ a'\ (X_v \cup X_v') \\ &=> \textit{In-Elem}\ a'\ X_v \qquad (From\ eq.2.3) \\ &=> \textit{In}\ a'^{(\textit{get-annot}\ a'\ (X_v \cup X_v'))}\ X_v \\ &=> \textit{In}\ a'\ \mathbb{X}[\![X_v]\!]_c \qquad (From\ eq.2.25)\end{aligned} \tag{2.27}$$

From Eq. 2.27 we get,

$$\mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c \equiv_{set} a' :: (\mathbb{X}[\![\textit{remove-elem}\ a'\ X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c) \tag{2.28}$$

Rewriting eq. 2.28 on the R.S. of eq. 2.26

$$a' :: \mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup X_v')]\!]_c \equiv_{set} a' :: (\mathbb{X}[\![\textit{remove-elem}\ a'\ X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c) \tag{2.29}$$

which is $True$ by induction hypothesis (Eq. 2.1).

Else if,

$$\mathbb{E}[\![\ \textit{get-annot}\ a'\ (X_v \cup X_v')]\!]_c = \textit{False} \tag{2.30}$$

from eq. 2.23 and 2.24 we get,

$$\mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup X_v')]\!]_c \equiv_{set}\ \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c \tag{2.31}$$

From definition of*get-annot* (Def. 2.3.4),

$$\begin{aligned}\mathbb{E}[\![\ \textit{get-annot}\ a'\ (X_v \cup X_v')]\!]_c = \textit{False} &=>\sim \textit{In-Elem}\ a'\ (X_v \cup X_v') \\ &=>\sim \textit{In-Elem}\ a'\ X_v \qquad (From\ eq.2.3) \\ &=>\sim \textit{In}\ a'\ \mathbb{X}[\![X_v]\!]_c\end{aligned} \tag{2.32}$$

From Eq. 2.32 we get,

$$\mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c \equiv_{set} \mathbb{X}[\![\textit{remove-elem}\ a'\ X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c \tag{2.33}$$

Rewriting eq. 2.33 on the R.S. of eq. 2.31,

$$\mathbb{X}[\![((\textit{remove-elem}\ a'\ X_v) \cup X_v')]\!]_c \equiv_{set} \mathbb{X}[\![\textit{remove-elem}\ a'\ X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c \tag{2.34}$$

which is $True$ by induction hypothesis (Eq. 2.1).

∗ case $\sim$ *In-Elem* a' X_v:

$$\sim \textit{In-Elem}\ \ a'\ X_v \tag{2.35}$$

From eq. 2.13 and 2.35, we get,

$$X_v \cup (a'^{e'} :: X_v') \equiv_{vset} a'^{e'} :: (X_v \cup X_v') \tag{2.36}$$

Rewriting above eq. 2.36 on the R.S. of our goal in eq. 2.2,

$$\mathbb{X}[\![a'^{e'} :: (X_v \cup : X_v')]\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![(a'^{e'} :: X_v')]\!]_c \tag{2.37}$$

From eq. 2.35 and 2.3, we can re-write eq.2.37 as,

$$\mathbb{X}[\![X_v \cup : X_v']\!]_c \equiv_{set} \mathbb{X}[\![X_v]\!]_c \cup \mathbb{X}[\![X_v']\!]_c \tag{2.38}$$

which is $True$ by induction hypothesis (Eq. 2.1).

Therefore, goal (Eq. 2.1) for inductive case is $True$ for all cases. ■

Formal proof of the above theorem is given below.

```
Theorem velems_union_is_variation_preserving : forall A  A' c (HA: NoDupElem A)
(HA': NoDupElem A'),
X[[ velems_union A A']]c =set= elems_union (X[[ A]] c) (X[[ A']] c).
Proof. intros A A'. generalize dependent A. induction A' as [|a' A' IHA'].
- (* case A' = [] *)
  (*
    ----------------------------------------------
    X[[ velems_union A []]] c =set= elems_union (X[[ A]] c) []
  *)
  destruct A as [| a A]; intros.
  + (* case A = []       *) simpl. reflexivity.
  + (* case A = (a :: A) *) simpl (X[[ _]] _) at 3.
    (* forall X, velems_union X [] =vset= [] ∧ forall x, elems_union x [] =set=
    [] *)
    rewrite velems_union_nil_r, elems_union_nil_r.
    reflexivity. assumption.

- (* case A' = (a :: A') *)
  intros A c Ha Ha'. destruct a' as (a', e').
  (* IHA' : X[[ velems_union A A']] c =set= elems_union (X[[ A]] c) (X[[ A']] c)
     Ha'  : NoDupElem (ae a' e' :: A')
     .........
     ----------------------------------------------
     X[[ velems_union A (ae a' e' :: A')]] c =set= elems_union (X[[ A]] c) (X[[
     ae a' e' :: A']] c)
  *)

  (* inversion NoDupElem (ae a' e' :: A') →    InElem a' A' *)
  inversion Ha' as [| a'' e'' A'' HnInElemA' HNdpElemA']; subst.
  simpl set_union.

  (*   InElem a' A' →    In (ae a' e') A'   *)
  pose (NoDupElem_NoDup Ha') as Hndp.
  inversion Hndp as [|a'' e'' HnInA' HNDpA']; subst. clear Hndp.

  (* IHA' : X[[ velems_union A A']] c =set= elems_union (X[[ A]] c) (X[[ A']] c)
     HnInA' :   In (ae a' e') A'
     HnInElemA' :   InElem a' A'
    .......
    ----------------------------------------------
    X[[ velems_union A (ae a' e' :: A')]] c =set= elems_union (X[[ A]] c) (X[[
    ae a' e' :: A']] c)
  *)

  (** Prove by cases of In (ae a' e') A  *)
  destruct (in_dec velem_eq_dec (ae a' e') A) as [HInA | HnInA].
  + (* case In (ae a' e') A *)
```

```
(* IHA' : X[[ velems_union A A']] c =set= elems_union (X[[ A]] c) (X[[ A']]
c)
   HnInA' :   In (ae a' e') A'
   HnInElemA' :   InElem a' A'
   HInA : In (ae a' e') A
   ........
   ----------------------------------------------
   X[[ velems_union A (ae a' e' :: A')]] c =set= elems_union (X[[ A]] c)
   (X[[
   ae a' e' :: A']] c)
*)

(* HInA : In (ae a' e') A → velems_union A (ae a' e' :: A') =vset=
velems_union A A' *)
apply velems_union_InA with (B:=A') in HInA as Hequiv.
unfold "=vset=" in Hequiv. rewrite Hequiv.
(* .......
  ----------------------------------------------
  X[[ velems_union A A']] c =set= elems_union (X[[ A]] c) (X[[ ae a' e' ::
  A']] c)
*)
simpl.
(* .......
  ----------------------------------------------
  X[[ velems_union A A']] c =set= elems_union (X[[ A]] c)
  (if E[[ e']] c then a' :: X[[ A']] c else X[[ A']] c)
*)
destruct (E[[ e']] c) eqn:He'.
++ (* case  (E[[ e']] c) = true *)
   (* HInA : In (ae a' e') A
   .......
   ----------------------------------------------
   X[[ velems_union A A']] c =set= elems_union (X[[ A]] c) (a' :: X[[ A']] c)
  *)
   simpl elems_union. unfold elems_union.
   unfold elems_union in IHA'.
   (* HInA : In (ae a' e') A
      .......
      ----------------------------------------------
      X[[ velems_union A A']] c =set= set_add string_eq_dec a' (elems_union
      (X[[ A]] c) (X[[ A']] c))
   *)
   (* HInA  : In (ae a' e') A → HInA_c: In a' (X[[A]]c) *)
   pose (In_config_true a' e' A c HInA He') as HInA_c.
   (* HInA_c: In a' (X[[A]]c) → In a' (elems_union (X[[ A]] c) (X[[ A']]
   c) *)
```

```
      apply (set_union_intro1 string_eq_dec) with (y:= (X[[ A']]c)) in
      HInA_c.
      (* HInA_c → set_add string_eq_dec a' (elems_union (X[[ A]] c) (X[[
      A']] c)) = elems_union (X[[ A]] c) (X[[ A']] c)  *)
      apply (In_set_add string_eq_dec) in HInA_c. rewrite HInA_c. clear
      HInA_c.

      (* IHA' : X[[ velems_union A A']] c =set= elems_union (X[[ A]]c) (X[[
      A']]c)
        .......
        -----------------------------------------------
        X[[ velems_union A A']] c =set= elems_union (X[[ A]] c) (X[[ A']] c)
      *)
      apply IHA'; eauto.

   ++ (* case  (E[[ e']] c) = false *)
      apply IHA'; eauto.

 + (* case   In (ae a' e') A *)

   (* IHA' : X[[ velems_union A A']] c =set= elems_union (X[[ A]] c) (X[[ A']]
   c)
      HnInA' :   In (ae a' e') A'
      HnInElemA' :   InElem a' A'
      .......
      -----------------------------------------------
      X[[ velems_union A (ae a' e' :: A')]] c =set= elems_union (X[[ A]] c)
      (X[[ ae a' e' :: A']] c)
   *)

   (** Proof by cases of existsbElem a' A *)
   destruct (existsbElem a' A) eqn:HexstElemA.
   ++ (* case existsbElem a' A = true → InElem a' A *)
      existsbElem_InElem in HexstElemA. rename HexstElemA into HInElemA.
      (* HnInA :   In (ae a' e') A
         HInElemA : InElem a' A
         ........
         -----------------------------------------------
         X[[ velems_union A (ae a' e' :: A')]] c =set= elems_union (X[[ A]] c)
         (X[[ ae a' e' :: A']] c)
      *)
      (* From velems_union Defn, HnInA ∧ HInElemA →
         velems_union A (ae a' e' :: A') =vset= ae a' (e' ∨ (F) extract_e a'
             (velems_union A A')):: velems_union (removeElem a' A) A']] c *)
      apply (velems_union_nInA_InElemA) with (A:=A) in Ha' as Hequiv;
      try( split; assumption); try assumption.
      unfold "=vset=" in Hequiv. rewrite Hequiv.
```

```
(* ........
----------------------------------------------
X[[ ae a' (e' ∨ (F) extract_e a' (set_union velem_eq_dec A A'))
:: velems_union (removeElem a' A) A']] c
=set= elems_union (X[[ A]] c) (X[[ ae a' e' :: A']] c)
*)
simpl.
(* ........
----------------------------------------------
(if (E[[ e']] c) || (E[[ extract_e a' (set_union velem_eq_dec A A')]]c)
  then a' :: X[[ velems_union (removeElem a' A) A']] c
   else X[[ velems_union (removeElem a' A) A']] c) =set= elems_union
    (X[[ A]] c) (if E[[ e']] c then a' :: X[[ A']] c else X[[ A']] c)
*)
(** Prove by cases (E[[ e']] c) *)
destruct (E[[ e']] c) eqn:He'.
+++ (* (E[[ e']] c) = true *)

    rewrite orb_true_l.
    (* HnInElemA' :   InElem a' A'
       .........
       ----------------------------------------------
       a' :: X[[ velems_union (removeElem a' A) A']] c
         =set= elems_union (X[[ A]] c) (a' :: X[[ A']] c)
    *)
    (* HnInElemA' :   InElem a' A' → elems_union (X[[ A]] c) (a' :: X[[
    A']] c)
    =set= a' :: elems_union (X[[ removeElem a' A]] c) (X[[ A']] c) *)
    rewrite (notInElemA'_set_union_cons_removeElem _ c Ha HNdpElemA'
    HnInElemA').
    apply cons_equiv_elems.

    (* IHA': X[[ velems_union A A']] c =set= elems_union (X[[ A]] c) (X[[
    A']] c)
       ..........
       ----------------------------------------------
       a' :: X[[ velems_union (removeElem a' A) A']] c
       =set= a' :: elems_union (X[[ removeElem a' A]] c) (X[[ A']] c)
    *)
    apply IHA'; eauto.

+++ (* (E[[ e']] c) = false *)
    rewrite orb_false_l.

    (* HnInElemA' :   InElem a' A'
      .........
      ----------------------------------------------
  (if E[[ extract_e a' (set_union velem_eq_dec A A')]] c
  then a' :: X[[ velems_union (removeElem a' A) A']] c
  else X[[ velems_union (removeElem a' A) A']] c) =set= elems_union
  (X[[ A]] c) (X[[ A']] c)
*)

(*   InElem a' A'→  E[[ extract_e a' (set_union velem_eq_dec A
A')]]c = E[[ extract_e a' A]]c *)
rewrite notInElemA'_extract_set_union; try assumption.
apply InElem_extract in HInElemA as HInAexe; try assumption.
destruct HInAexe as [e [HInA Hexeqe] ].

(* .........
   -----------------------------------------------
   (if E[[ extract_e a' A]] c
   then a' :: X[[ velems_union (removeElem a' A) A']] c
   else X[[ velems_union (removeElem a' A) A']] c) =set= elems_union
   (X[[ A]] c)
   (X[[ A']] c)
*)

(** Prove by cases (E[[ e']] c) *)
destruct (E[[ extract_e a' A]] c) eqn: Hexta'.

++++ (* E[[ extract_e a' A]]c = true *)

     (* E[[ extract_e a' A]]c = true →  E[[e]]c = true *)
     rewrite Hexeqe in Hexta'. simpl in Hexta'.
     rewrite orb_false_r in Hexta'.

     (* E[[e]]c = true ∧  In (ae a e) A →  In a X[[A]]c *)
     apply In_config_true with (c:=c) in HInA; try assumption.

     (* In a X[[A]]c →  elems_union (X[[ A]] c) (X[[ A']] c) =set=
     (a' :: elems_union (X[[ removeElem a' A]] c) (X[[ A']] c)) *)
     rewrite (In_set_union_removeElem _ c Ha HNdpElemA' HInA
     HnInElemA').
     apply cons_equiv_elems.

     apply IHA'; eauto.

++++ (* E[[ extract_e a' A]]c = false *)

     (* E[[ extract_e a' A]]c = false →  E[[e]]c = false *)
     rewrite Hexeqe in Hexta'. simpl in Hexta'.
     rewrite orb_false_r in Hexta'.
```

```
            (*  E[[e]]c = false ∧  In (ae a e) A →    In a X[[[A]]c *)
            apply In_config_false with (c:=c) in HInA; try assumption.

            (*   In a X[[A]]c →  elems_union (X[[ A]] c) (X[[ A']] c) =set=
            elems_union (X[[ removeElem a' A]] c) (X[[ A']] c) *)
            rewrite (notInElem_set_union_removeElem _ c Ha HNdpElemA' HInA
            HnInElemA').

            apply IHA'; eauto.

    ++ (* case existsbElem a' A = false →    InElem a' A *)
    not_existsbElem_InElem in HexstElemA.
    rename HexstElemA into HnInElemA.

    (*  HnInA    :   In (ae a' e') A
    HnInElemA :   InElem a' A
    ........
    -----------------------------------------------
    X[[ velems_union A (ae a' e' :: A')]] c =set= elems_union (X[[ A]] c) (X[[
    ae a'
    e' :: A']] c)
    *)
    (* HnInA ∧  HnInElemA →
    velems_union A (ae a' e' :: A') =vset= ae a' e' :: velems_union A A' *)
    apply (velems_union_nInA_nInElemA) with (A:=A) in Ha' as Hequiv;
    try(split; assumption).
    unfold "=vset=" in Hequiv. rewrite Hequiv.
    (* HnInElemA' :   InElem a' A'
       HnInElemA  :   InElem a' A
       ..............
       -----------------------------------------------
       X[[ ae a' e' :: velems_union A A']] c =set= elems_union (X[[ A]] c) (X[[
       ae a' e' :: A']] c)
    *)
   (* Goal →  HnInElemA' ∧  HnInElemA ∧  IHA'  *)
   apply velems_union_nInElemA_nInElemB; eauto.

Qed.
```

A.2.4 Plain Set Intersection of Empty Set

```
(* Plain set intersection nil-r *)
Lemma elems_inter_nil_r: forall A, elems_inter A [] = [].
Proof. intro A. induction A; eauto. Qed.
```

```
(* Plain set intersection nil-l *)
Lemma elems_inter_nil_l: forall A, elems_inter [] A = [].
Proof. eauto. Qed.
```

A.2.5 V-Set Intersection of Empty Set

```
(* V-set intersection nil-r *)
Lemma velems_inter_nil_r : forall A, velems_inter A [] = [].
Proof. intro A. induction A as [|(a, e)]. reflexivity.
rewrite velems_inter_equation. simpl.
assumption. Qed.

(* V-set intersection nil-l *)
Lemma velems_inter_nil_l : forall A, velems_inter [] A = [].
Proof. intros. rewrite velems_inter_equation. simpl. reflexivity. Qed.
```

A.2.6 V-Set Intersection Variation Preservation

```
Theorem velems_intersection_is_variation_preserving : forall A  A' c (HA: NoDupElem
 A) (HA': NoDupElem A'),
           X[[ velems_inter A A']] c = elems_inter (X[[ A]] c) (X[[ A']] c).
  Proof. intros. induction A as [|a A IHA].
  - (* case A = [] *) simpl. reflexivity.
  - (* case A = (a::A) *)
     simpl.  destruct a as (a, e).
     (* get (  InElemA a A) from NoDupElem (ae a e :: A) *)
     inversion HA as [| a'' e'' A'' HnInElemA HNdpElemA]; subst.
     (* rewrite velems_inter equation*)
     rewrite velems_inter_equation.
     (** Proof by cases of (E[[ e]]c) *)
     destruct (E[[ e]]c) eqn:He.
     { (* case He: (E[[e]]c) = true *)
        simpl elems_inter.
       (** Proof by cases of (set_mem _ a (X[[ A']] c)) *)
       destruct (set_mem string_eq_dec a (X[[ A']] c)) eqn: Hset_memaA'.
      + (* case (set_mem _ a (X[[ A']] c) = true *)
        (* set_mem _ a (X[[ A']] c) = true → In a (X[[ A']] c) *)
        apply (set_mem_correct1 string_eq_dec) in Hset_memaA'.
        (* In a (X[[ A']] c) → Hget_annot: E[[ get_annot a A']] c = true *)
        apply get_annot_true_In in Hset_memaA' as Hget_annot.
        (* In a (X[[ A']] c) → HInelemaA': InElem a A' *)
        apply In_InElem_config in Hset_memaA' as HInelemaA'.
        (* InElem a A' → existsbElem a A' = true *)
        rewrite ← existsbElem_InElem in HInelemaA'. rewrite HInelemaA'.
        (* simpl X[[_]]c. rewrite He Hget_annot IHA *)
        simpl configVElemSet. rewrite He, Hget_annot, IHA. simpl.
        reflexivity.  assumption.
      + (* case (set_mem _ a (X[[ A']] c) <> true *)
        (* set_mem _ a (X[[ A']] c) <> true →    In a (X[[ A']] c) *)
        apply (set_mem_complete1 string_eq_dec) in Hset_memaA'.
        (*   In a (X[[ A']] c) → Hget_annot: E[[ get_annot a A']] c = false *)
        rewrite ← get_annot_true_In in Hset_memaA'.
        (* rewrite <> true ↔  = false in Hset_memaA' *)
        rewrite not_true_iff_false in Hset_memaA'.
        (** Proof by cases of existsbElem a A' *)
        destruct (existsbElem a A').
        ++ (* existsbElem a A' = true *)
           (* simpl X[[_]]c. rewrite IHA Hset_memaA' *)
           simpl configVElemSet. rewrite IHA, Hset_memaA'.
           rewrite andb_false_r. reflexivity. assumption.
        ++ (* existsbElem a A' = false *) apply(IHA HNdpElemA).
    }
    { (* case He: (E[[e]]c) = true *)
      (** Proof by cases of existsbElem a A' *)
      destruct (existsbElem a A').
      + (* existsbElem a A' = true *)
        (* simpl X[[_]]c. rewrite He IHA *)
        simpl configVElemSet. rewrite He, IHA.
        rewrite andb_false_l. reflexivity. assumption.
      + (* existsbElem a A' = false *) apply(IHA HNdpElemA).
    }
  Qed.
```

A.2.7 Annotated V-Set Union Variation Preservation

```
Theorem avelems_union_vq_is_variation_preserving : forall Q Q' c (HA: NoDupElem
(fst Q))(HA': NoDupElem (fst Q')),
      AX[[(vqtype_union_vq Q Q')]]c =set= elems_union (AX[[ Q]]c) (AX[[ Q']]c).
Proof.
intros Q Q' c HQ HQ'.
destruct Q as (A, e). destruct Q' as (A', e').
unfold vqtype_union_vq, configaVelems.
simpl fst. simpl snd. simpl.
destruct (E[[ e]] c) eqn: He; simpl;
[ | destruct (E[[ e']] c) eqn: He'; simpl;
[ | (* [] =set= [] *)simpl; reflexivity] ];
rewrite configVElemSet_dist_velems_union;
try (apply NoDupElem_push_annot; auto); simpl;
```

```
repeat(rewrite configVElemSet_push_annot); simpl;
rewrite He; [|rewrite He']; reflexivity.
Qed.
```

A.2.8 Annotated V-Set Intersection Variation Preservation

```
Theorem avelems_intersection_vq_is_variation_preserving : forall Q  Q' c (HQ:
NoDupElem (fst Q)) (HQ': NoDupElem (fst Q')),
     AX[[ avelems_inter_vq Q Q']] c = elems_inter (AX[[ Q]] c) (AX[[ Q']] c).
Proof.
intros Q Q' c HQ HQ'.
destruct Q as (A, e). destruct Q' as (A', e').
unfold avelems_inter_vq. simpl. simpl in *.
destruct (E[[ e]] c) eqn: He;
destruct (E[[ e']] c) eqn: He'; simpl; try reflexivity.
+ apply velems_intersection_is_variation_preserving; auto.
+ rewrite elems_inter_nil_r. auto.
Qed.
```

Appendix B: Formal Encoding of Variational Query

B.1 RA Type System

Plain Query Type:

$$. \mid\mid= \; . \qquad : \mathbf{Q} \rightarrow \mathbf{S} \rightarrow \mathbf{QT}$$

$$S \mid\mid= \; r(A) \quad = \begin{cases} A, \text{ if } r \in S \\ \{\}, \text{ otherwise} \end{cases}$$

$$S \mid\mid= \; \pi_A q \quad = \begin{cases} A, \text{ if } A \subseteq (S \mid\mid= \; q) \\ \{\}, \text{ otherwise} \end{cases}$$

$$S \mid\mid= \; \sigma_\theta q \quad = \begin{cases} S \mid\mid= \; q, \text{ if } (S \mid\mid= \; q) \mid\mid= \; \theta \\ \{\}, \text{ otherwise} \end{cases}$$

$$S \mid\mid= \; q_1 \times q_2 = \begin{cases} (S \mid\mid= \; q_1) \cup (S \mid\mid= \; q_2), \text{ if } (S \mid\mid= \; q_1) \cap (S \mid\mid= \; q_2) = \{\} \\ \{\}, \text{ otherwise} \end{cases}$$

$$S \mid\mid= \; q_1 \circ q_2 = \begin{cases} (S \mid\mid= \; q_1), \text{ if } (S \mid\mid= \; q_1) \equiv_{set} (S \mid\mid= \; q_2) \\ \{\}, \text{ otherwise} \end{cases}$$

$$S \mid\mid= \; \varepsilon \quad = \{\}$$

Plain query type function is encoded as `type_` in Coq.

```
(* ------------------------------------------------------------
| Type of plain query
------------------------------------------------------------- *)
Fixpoint type_ (q:query) (s:schema) : qtype :=
match q with
| (rel (rn, A))   ⇒  if (existsb (relS_beq (rn, A)) s) then A else []
| (proj A q)      ⇒  let A' := type_ q s in
                     if subset_qtype_bool A A' then A else []
| (sel c q)       ⇒  let A := type_ q s in
                     if (condtype c A) then A else []
```

```
| (setU op q1 q2) ⇒  if equiv_qtype_bool (type_ q1 s) (type_ q2 s) then type_ q1
                                                  else []
| (prod  q1 q2)   ⇒  if (is_disjoint_bool (type_ q1 s) (type_ q2 s)) then
                  elems_union (type_ q1 s) (type_ q2 s) else []
| (empty)         ⇒  []
end.

Notation "s ||= q " := (type_ q s) (at level 49).
```

Plain Condition Type Check:

$$
\begin{aligned}
. \mid\mid = \ . & \quad : \boldsymbol{\Theta} \rightarrow \mathbf{A} \rightarrow \mathbf{B} \\
A \mid\mid = \ b & = \texttt{true} \\
A \mid\mid = \ a \bullet k & = \texttt{true} \\
A \mid\mid = \ a_1 \bullet a_2 & = \texttt{true} \\
A \mid\mid = \ \neg\theta & = \neg(A \mid\mid = \ \theta) \\
A \mid\mid = \ \theta_1 \vee \theta_2 & = (A \mid\mid = \ \theta_1) \vee (A \mid\mid = \ \theta_2) \\
A \mid\mid = \ \theta_1 \wedge \theta_2 & = (A \mid\mid = \ \theta_1) \wedge (A \mid\mid = \ \theta_2) \\
A \mid\mid = \ e\langle\theta_1, \theta_2\rangle & = \begin{cases} A \mid\mid = \ \theta_1, & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ A \mid\mid = \ \theta_2, & \text{otherwise} \end{cases}
\end{aligned}
$$

Plain condition type check function is encoded as `condtype` in Coq.

```
(* ------------------------------------------------------------
| Type check of plain condition
------------------------------------------------------------ *)
Fixpoint condtype (c:cond) (A:elems) : bool :=
match c with
| litCB b         ⇒  true
| elemOpV o a n   ⇒  true
| elemOpA o a1 a2 ⇒  true
| negC  c     ⇒  if (condtype c  A)                    then true else false
| conjC c1 c2 ⇒  if (condtype c1 A) && (condtype c2 A) then true else false
| disjC c1 c2 ⇒  if (condtype c1 A) && (condtype c2 A) then true else false
end.
Notation "A ||- c " := (condtype c A) (at level 49).

(*-----------------------------type'-----------------------------*)
```

B.2 Formal Proof of Correctness of VRA Type System

```
Theorem variation_preservation : forall e S vq A' e',
        { e , S |= vq | (A', e') } →
        forall c, E[[e]]c = true →
           (S[[ S]]c) ||= (Q[[ vq]]c) =set= QT[[ (A', e')]]c.
 Proof.
   intros e S vq A'' e'' H c H0.
    induction H as [
                    |
                    e S HndpRS HndpAS
                    rn HeR A' HndpA' e'
                    HInVR
                    |
                    e S HndpRS HndpAS vq HndpvQ
                    e' A' HndpAA' Q HndpQ
                    Hq IHHq Hsbsmp
                    |
                    e S HndpRS HndpAS
                    vq HndpvQ A HndpAA e' vc
                    Hq IHHq HCond

                    |
                    e e' S HndpRS HndpAS
                    vq1 HndpvQ1 vq2 HndpvQ2
                    A1 HndpAA1 e1 A2 HndpAA2 e2
                    Hq1 IHHq1 Hq2 IHHq2
                    |
                    e S HndpRS HndpAS
                    vq1 HndpvQ1 vq2 HndpvQ2
                    A1 HndpAA1 e1 A2 HndpAA2 e2
                    Hq1 IHHq1 Hq2 IHHq2 HInter
                    |
                    e S HndpRS HndpAS
                    vq1 HndpvQ1 vq2 HndpvQ2
                    A1 HndpAA1 e1 A2 HndpAA2 e2 op
                    Hq1 IHHq1 Hq2 IHHq2 HEquiv
                    ].
  (** ----------------------------- EmptyRelation - E -----------------------------
  *)
   -
   (* H0 : (E[[ e]] c) = true
      ----------------------------------------------
      ||= (Q[[ empty_v]] c) =set= QT[[ ([], litB false)]] c
   *)
   unfold configVQuery, configVQtype, configaVelems. simpl. reflexivity.
```

```
(** ----------------------------- Relation - E ----------------------------- *)
 -
 (* H0 : (E[[ e]] c) = true
    -----------------------------------------------
    ||= (Q[[ rel_v (rn, (A', e'))]] c) =set= (QT[[ (A', e ∧ (F) e')]] c)
 *)
 unfold configVQuery, configVQtype, configaVelems. simpl semE.
 (* H0 : (E[[ e]] c) = true
    -----------------------------------------------
    ||= rel (R[[ (rn, (A', e'))]] c) =set=
                     (if (E[[ e]] c) && (E[[ e']] c) then X[[ A']] c else [])
 *)
 (* (E[[ e]] c) = true *)
 rewrite H0. rewrite andb_true_l.
 (* Proved by definitions InVR_In, configVRelS and ||= rel (rn, A) = A*)
 rewrite type__configVRelS. apply InVR_In with (c:=c) in HInVR; try auto.
 unfold configVRelS in HInVR. simpl in HInVR.
 rewrite ← existsb_In_relS in HInVR. destruct (E[[e']]c).
 rewrite HInVR. all: reflexivity.

(** ----------------------------- Project - E ----------------------------- *)
 -
 (* Hq: {e, S |= vq | (A', e')}
    Hsbsmp: subset_vqtype (Q ^^ e) (A', e')
    -----------------------------------------
    ||= (Q[[ proj_v Q vq]] c) =set= (QT[[ Q ^^ e]] c)
 *)
 (*  unfold ||=. AE = QT. Simplify IHHq with (E[[ e]] c) = true.  *)
 simpl type_. rewrite AX_QT. unfold subset_qtype_bool. apply IHHq in H0 as IHHq'.
 clear IHHq.
 (* Hq: {e, S |= vq | (A', e')}
    Hsbsmp: subset_vqtype (Q ^^ e) (A', e')
    IHHq' : ||= (Q[[ vq]] c) =set= (QT[[ (A', e')]] c)
    -----------------------------------------
    if subset_bool (QT[[ Q]] c) (||= (Q[[ vq]] c)) then
         QT[[ Q]] c  =set= (QT[[ Q ^^ e]] c)
 *)
 (* rewrite IHHq' in goal *)
 rewrite (equiv_subset_bool _ IHHq').
 (* Hq: {e, S |= vq | (A', e')}
    Hsbsmp: subset_vqtype (Q ^^ e) (A', e')
    -----------------------------------------
    if subset_bool (QT[[ Q]] c) (QT[[ (A', e')]] c) then
         QT[[ Q]] c  =set= (QT[[ Q ^^ e]] c)
 *)
 (*  By defintion, subset_vqtype A B = subset QT[[A]] QT[[B]]  *)
 unfold subset_vqtype in Hsbsmp. specialize Hsbsmp with c.
```

```
(* Hq: {e, S |= vq | (A', e')}
   Hsbsmp : subset (QT[[ Q ^^ e]] c) (QT[[ (A', e')]] c)
   ----------------------------------------
   if subset_bool (QT[[ Q]] c) (QT[[ (A', e')]] c) then
       QT[[ Q]] c  =set= (QT[[ Q ^^ e]] c)
*)
(*  (E[[ e]] c) = true → (QT[[ Q ^^ e]] c) = (QT[[ Q]] c)   *)
rewrite (addannot_config_true _ _ _ H0) in Hsbsmp. rewrite (addannot_config_true
_ _ _ H0).
(* Hq: {e, S |= vq | (A', e')}
   Hsbsmp : subset (QT[[ Q]] c) (QT[[ (A', e')]] c)
   ----------------------------------------
   if subset_bool (QT[[ Q]] c) (QT[[ (A', e')]] c) then
       QT[[ Q]] c  =set= (QT[[ Q]] c)
*)
(*  Proved by subset A B ↔ subset_bool A B = true  *)
rewrite ← subset_bool_correct in Hsbsmp. rewrite Hsbsmp.
reflexivity.

(**  ------------------------------- Select - E ------------------------------ *)
- apply IHHq in H0 as Htype_.
  simpl configVQuery.
  simpl type_.

(* HCond : {e, (A, e') |- vc}
   Htype_ : ||= (Q[[ vq]] c) =set= (QT[[ (A, e')]] c)
   -----------------------------------------------------
   (if (QT[[ (A, e')]] c) ||- (C[[ vc]] c)
              then ||= (Q[[ vq]] c) else []) =set= (QT[[ (A, e')]] c)
*)

(* {e, (A, e') |- vc} →  (QT[[ (A, e')]] c) ||- (C[[ vc]] c) = true *)

apply variation_preservation_cond with (c:=c) in HCond.

(* HCond : (QT[[ (A, e')]] c) ||- (C[[ vc]] c) = true
   Htype_ : ||= (Q[[ vq]] c) =set= (QT[[ (A, e')]] c)
   -----------------------------------------------------
   (if (||= (Q[[ vq]] c)) ||- (C[[ vc]] c)
                 then ||= (Q[[ vq]] c) else []) =set= (QT[[ (A, e')]] c)
*)

(*  v-condition (C[[ vc]] c) is well formed in all equivalent contexts:
    Htype_:      ||= (Q[[ vq]] c) =set= (QT[[ (A, e')]] c) →
    HCond_:  ||= (Q[[ vq]] c) ||- (C[[ vc]] c) = (QT[[ (A, e')]] c) ||- (C[[
    vc]] c)  *)
```

```
 apply condtype_equiv with (c:=(C[[ vc]] c)) in Htype_ as HCond_.

(* HCond : (QT[[ (A, e')]] c) ||- (C[[ vc]] c) = true
   Htype_ : ||= (Q[[ vq]] c) =set= (QT[[ (A, e')]] c)
   HCond_ : (||= (Q[[ vq]] c)) ||- (C[[ vc]] c) = (QT[[ (A, e')]] c) ||- (C[[
   vc]] c)
   ----------------------------------------------------
   (if (||= (Q[[ vq]] c)) ||- (C[[ vc]] c)
                        then ||= (Q[[ vq]] c) else []) =set= (QT[[ (A, e')]] c)
 *)

 rewrite HCond_, HCond. assumption. auto.

(**  ------------------------------ Choice - E ------------------------------ *)
-
(* Hq1 : {e ∧ (F) e', S |= vq1 | (A1, e1)}
   Hq2 : {e ∧ (F)  (F) e', S |= vq2 | (A2, e2)}
   H0 : (E[[ e]] c) = true
   IHHq1 : (E[[ e ∧ (F) e']] c) = true →  ||= (Q[[ vq1]] c) =set= (QT[[ (A1,
   e1)]] c)
   IHHq2 : (E[[ e ∧ (F)  (F) e']] c) = true →  ||= (Q[[ vq2]] c) =set= (QT[[ (A2,
   e2)]] c)
   ----------------------------------------------------------
   ||= (Q[[ chcQ e' vq1 vq2]] c) =set= (QT[[ vqtype_union_vq (A1, e1) (A2, e2)]]
   c)
*)
(*  Hq1 Hq2: contex_typeannot_rel →  {e, _ |= _ | (_, e')} →  ( e →   e')   *)
   apply context_type_rel in Hq1. rewrite not_sat_not_prop, ←  sat_taut_comp_inv
   in Hq1.
   apply context_type_rel in Hq2. rewrite not_sat_not_prop, ←  sat_taut_comp_inv
   in Hq2.
   specialize Hq1 with c. specialize Hq2 with c.
(*  remove e from Hypotheses with (E[[ e]] c) = true  *)
   simpl semE in *. rewrite H0 in *. rewrite andb_true_l in *. Search negb.
   rewrite negb_false_iff in Hq2. rewrite negb_true_iff in IHHq2.
(* Hq1 :  (E[[ e']] c) = false →  (E[[ e1]] c) = false
   Hq2 :  (E[[ e']] c) = true  →  (E[[ e2]] c) = false
   IHHq1 :(E[[ e']] c) = true  →  ||= (Q[[ vq1]] c) =set= (QT[[ (A1, e1)]] c)
   IHHq2 :(E[[ e']] c) = false →  ||= (Q[[ vq2]] c) =set= (QT[[ (A2, e2)]] c)
   ----------------------------------------------------------
   ||= (Q[[ chcQ e' vq1 vq2]] c) =set= (QT[[ vqtype_union_vq (A1, e1) (A2, e2)]]
   c)
*)
(*  (Q[[ chcQ e' vq1 vq2]] c) →  (if E[[ e']] c then Q[[ vq1]] c else Q[[ vq2]]
c)  *)
   simpl configVQuery.
(*  (QT[[ vqtype_union_vq A B]] c) =set= elems_union (QT[[A]] c) (QT[[B]] c) *)
```

```
   rewrite configVQType_dist_vqtype_union_vq; try assumption.
   repeat (rewrite configVQType_push_annot).
(* (E[[ e']] c) = true →
   (E[[ e2]] c) = false →  elems_union (QT[[ (A1, e1)]] c) (QT[[ (A2, e2)]] c) =
   (QT[[ (A1, e1)]] c)  *)
   assert(Hq1': (E[[ e']] c) = true →  elems_union (QT[[ (A1, e1)]] c) (QT[[ (A2,
   e2)]] c) = (QT[[ (A1, e1)]] c)).
   intro. apply Hq2 in H. simpl. rewrite H. eauto.
(* (E[[ e']] c) = false →
   (E[[ e1]] c) = false →  elems_union (QT[[ (A1, e1)]] c) (QT[[ (A2, e2)]] c) =
   (QT[[ (A2, e2)]] c)  *)
   assert(Hq2': (E[[ e']] c) = false →  elems_union (QT[[ (A1, e1)]] c) (QT[[
   (A2, e2)]] c) =set= (QT[[ (A2, e2)]] c)).
   intro. apply Hq1 in H. simpl. rewrite H. rewrite elems_union_nil_l.
   reflexivity.
   destruct ( E[[ e2]] c); [ apply NoDupElem_NoDup_config | apply NoDup_nil];
   auto.
(* IHHq1 :(E[[ e']] c) = true  →  ||= (Q[[ vq1]] c) =set= (QT[[ (A1, e1)]] c)
   IHHq2 :(E[[ e']] c) = false →  ||= (Q[[ vq2]] c) =set= (QT[[ (A2, e2)]] c)
   Hq1' : (E[[ e']] c) = true →
      elems_union (QT[[ (A1, e1)]] c) (QT[[ (A2, e2)]] c) = (QT[[ (A1, e1)]] c)
   Hq2' : (E[[ e']] c) = false →
      elems_union (QT[[ (A1, e1)]] c) (QT[[ (A2, e2)]] c) =set= (QT[[ (A2, e2)]]
      c)
   ----------------------------------------------------------
   ||= (if E[[ e']] c then Q[[ vq1]] c else Q[[ vq2]] c) =set= elems_union (QT[[
   (A1, e1)]] c) (QT[[ (A2, e2)]] c)
*)

   destruct (E[[ e']] c) eqn: He'.

(* He'   :(E[[ e']] c) = true
   IHHq1 :||= (Q[[ vq1]] c) =set= (QT[[ (A1, e1)]] c)
   Hq1'  :elems_union (QT[[ (A1, e1)]] c) (QT[[ (A2, e2)]] c) = (QT[[ (A1, e1)]]
   c)
   ----------------------------------------------------------
   ||= (Q[[ vq1]] c) =set= elems_union (QT[[ (A1, e1)]] c) (QT[[ (A2, e2)]] c)
*)
   rewrite Hq1'; try reflexivity. apply IHHq1; try reflexivity.
(* He'   :(E[[ e']] c) = false
   IHHq2 : ||= (Q[[ vq2]] c) =set= (QT[[ (A2, e2)]] c)
   Hq2'  : elems_union (QT[[ (A1, e1)]] c) (QT[[ (A2, e2)]] c) =set= (QT[[ (A2,
   e2)]] c)
   ----------------------------------------------------------
   ||= (Q[[ vq1]] c) =set= elems_union (QT[[ (A1, e1)]] c) (QT[[ (A2, e2)]] c)
*)
   rewrite Hq2'; try reflexivity. apply IHHq2; try reflexivity.
```

```
(**  ------------------------------- Product - E ------------------------------ *)
 -
(* HInter : velems_inter A1 A2 =vset= []
   H0 : (E[[ e]] c) = true
   IHHq1 : (E[[ e]] c) = true → ||= (Q[[ vq1]] c) =set= (QT[[ (A1, e1)]] c)
   IHHq2 : (E[[ e]] c) = true → ||= (Q[[ vq2]] c) =set= (QT[[ (A2, e2)]] c)
   ----------------------------------------------------------
   ||= (Q[[ prod_v vq1 vq2]] c) =set= (QT[[ vqtype_union_vq (A1, e1) (A2, e2)]] c)
*)
(*  apply E[[ e]] c) = true in Inductive H  *)
   apply IHHq2 in H0 as IHHq2'. apply IHHq1 in H0 as IHHq1'.
   clear IHHq1. clear IHHq2.
(*  (QT[[ vqtype_union_vq A B]] c) =set= elems_union (QT[[A]] c) (QT[[B]] c) *)
   rewrite configVQType_dist_vqtype_union_vq; try assumption.
   repeat (rewrite configVQType_push_annot).
(*  ||= (Q[[ prod_v vq1 vq2]] c) ||= prod (Q[[ vq1]] c) (Q[[ vq2]] c) *)
   simpl configVQuery.
(*
  ----------------------------------------------------
  ||= prod (Q[[ vq1]] c) (Q[[ vq2]] c) =set= elems_union (QT[[ (A1, e1)]] c)
  (QT[[ (A2, e2)]] c)
*)
   simpl type_.
(* HInter : velems_inter A1 A2 =vset= []
   ----------------------------------------------------
   if is_disjoint_bool (||= (Q[[ vq1]] c)) (||= (Q[[ vq2]] c))
    elems_union (||= (Q[[ vq1]] c)) (||= (Q[[ vq2]] c)) =set= elems_union (QT[[
    (A1, e1)]] c) (QT[[ (A2, e2)]] c)
*)

(*  velems_inter A1 A2 =vset= [] →  elems_inter [[A1]]c [[A2]]c =set= []  *)
   unfold equiv_velems in HInter. unfold vqtype_inter_vq, " =vqtype=", "=avset="
   in
   HInter.
   specialize HInter with c. simpl in HInter.
   rewrite configVElemSet_dist_velems_inter in HInter; try assumption.
   assert (HInter': elems_inter (QT[[ (A1, e1)]] c) (QT[[ (A2, e2)]] c) =set= []
   ).
   simpl. destruct (E[[ e1]] c); [ destruct (E[[ e2]] c); [ assumption |
   rewrite elems_inter_nil_r; reflexivity] | rewrite elems_inter_nil_l;
   reflexivity ].
(* HInter' : elems_inter (QT[[ (A1, e1)]] c) (QT[[ (A2, e2)]] c) =set= []
   ----------------------------------------------------
   if is_disjoint_bool (||= (Q[[ vq1]] c)) (||= (Q[[ vq2]] c))
    elems_union (||= (Q[[ vq1]] c)) (||= (Q[[ vq2]] c)) =set= elems_union (QT[[
    (A1, e1)]] c) (QT[[ (A2, e2)]] c)
*)

(*  is_disjoint_bool A B = true → elems_inter A B = []  *)
   rewrite (is_disjoint_bool_equiv) with (B := (QT[[(A1, e1)]]c)) (B':= (QT[[(A2,
   e2)]] c)); try assumption.
   apply nil_equiv_eq in HInter'. rewrite ← is_disjoint_bool_correct in HInter'.
   rewrite HInter'.
(* IHHq1' : ||= (Q[[ vq1]] c) =set= (QT[[ (A1, e1)]] c)
   IHHq2' : ||= (Q[[ vq2]] c) =set= (QT[[ (A2, e2)]] c)
   ----------------------------------------------------
   elems_union (||= (Q[[ vq1]] c)) (||= (Q[[ vq2]] c)) =set= elems_union (QT[[
   (A1, e1)]] c) (QT[[ (A2, e2)]] c)
*)
(*  Proved by set_union_quiv  *)
  rewrite (set_union_equiv) with (B := (QT[[(A1, e1)]]c)) (B':= (QT[[(A2,
  e2)]]c)); try (eauto; reflexivity).
(*  NoDup assumptions  *)
   1, 2, 5, 6: try(apply (NoDup_equiv_elems IHHq1')); try(apply
   (NoDup_equiv_elems IHHq2')).

   1, 3, 5, 7   : simpl; destruct ( E[[ e1]] c).
   9, 10, 11, 12: simpl; destruct ( E[[ e2]] c).
   all: try(apply NoDupElem_NoDup_config; auto); try (apply NoDup_nil).

(**  ------------------------------- SetOp - E ------------------------------ *)
-
(* HEquiv : (A1, e1)  =vqtype= (A2, e2)
   H0 : (E[[ e]] c) = true
   IHHq1 : (E[[ e]] c) = true → ||= (Q[[ vq1]] c) =set= (QT[[ (A1, e1)]] c)
   IHHq2 : (E[[ e]] c) = true → ||= (Q[[ vq2]] c) =set= (QT[[ (A2, e2)]] c)
   ----------------------------------------------------------
   ||= (Q[[ setU_v op vq1 vq2]] c) =set= (QT[[ (A1, e1)]] c)
*)
(*  apply E[[ e]] c) = true in Inductive H  *)
   apply IHHq2 in H0 as IHHq2'. apply IHHq1 in H0 as IHHq1'.
   clear IHHq1. clear IHHq2.

(*  ||= (Q[[ setU_v vq1 vq2]] c) ||= prod (Q[[ vq1]] c) (Q[[ vq2]] c) *)
   simpl configVQuery.
(* IHHq1' : ||= (Q[[ vq1]] c) =set= (QT[[ (A1, e1)]] c)
   IHHq2' : ||= (Q[[ vq2]] c) =set= (QT[[ (A2, e2)]] c)
   ----------------------------------------------------
   ||= setU op (Q[[ vq1]] c) (Q[[ vq2]] c) =set= (QT[[ (A1, e1)]] c)
*)
   simpl type_.
(* HEquiv : (A1, e1)  =vqtype= (A2, e2)
    ----------------------------------------------------
    if equiv_qtype_bool (||= (Q[[ vq1]] c)) (||= (Q[[ vq2]] c))
               then ||= (Q[[ vq1]] c) =set= (QT[[ (A1, e1)]] c)
 *)
 (*   (A1, e1)  =vqtype= (A2, e2) →  (QT[[ (A1, e1)]] c) =set= (QT[[ (A2, e2)]] c)
  *)
  apply configVQtype_equiv with (c:=c) in HEquiv. rewrite ← IHHq1', ← IHHq2' in
  HEquiv.
 (*  Proved by A =set= B →  equiv_qtype_bool A B = true  *)
  rewrite ←  equiv_qtype_bool_correct in HEquiv. rewrite HEquiv. assumption.


Qed.
```

Appendix C: Formal Encoding of Implicitly Annotated Variational Query

C.1 Implicit VRA Type System

```
(* ---------------------------------------------------------------
   | Type of (Implicit |- ) variational query
    ---------------------------------------------------------------
*)
 Inductive vtypeImp :fexp → vschema → vquery → vqtype → Prop :=
   (*   -- EMPTYRELATION-E --  *)
   | EmptyRelation_vE_imp : forall e S {HndpRS:NoDupRn (fst S)}
                                    {HndpAS: NODupElemRs S},
        vtypeImp e S (empty_v) ([], litB false)
   (*   -- RELATION-E --  *)
   | Relation_vE_imp : forall e S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                            rn {Hrn: empRelInempS rn} A_ A' {HndpA': NoDupElem
                            A'} e_ e',
        InVR (rn, (A', e')) S →
         sat (e ∧ (F) e') →
          vtypeImp e S (rel_v (rn, (A_, e_))) (A', (e ∧ (F) e'))
   (*   -- PROJECT-E --  *)
   | Project_vE_imp: forall e S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                          vq {HndpvQ: NoDupElemvQ vq} e' A'
                            {HndpAA': NoDupElem A'} Q {HndpQ: NoDupElem (fst
                            Q)},
        vtypeImp e S vq (A', e') →
         subsump_vqtype Q (A', e') →
          vtypeImp e S (proj_v Q vq) (vqtype_inter_vq Q (A', e'))
   (*  -- SELECT-E --  *)
   | Select_vE_imp: forall e S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                         vq {HndpvQ: NoDupElemvQ vq}
                          A {HndpAA: NoDupElem A} e' vc,
        vtypeImp e S vq (A, e') →
         { e, (A, e') |- vc } →
          vtypeImp e S (sel_v vc vq) (A, e')
   (*  -- CHOICE-E --  *)
   | Choice_vE_imp: forall e e' S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                           vq1 {HndpvQ1: NoDupElemvQ vq1} vq2 {HndpvQ2:
                           NoDupElemvQ vq2}
```

```
                              A1 {HndpAA1: NoDupElem A1} e1 A2 {HndpAA2: NoDupElem
                              A2} e2,
        vtypeImp (e ∧ (F) e') S vq1 (A1, e1) →
         vtypeImp (e ∧ (F) ( (F) e')) S vq2 (A2, e2) →
          vtypeImp e S (chcQ e' vq1 vq2)
           (vqtype_union_vq (A1, e1) (A2, e2))
   (*  -- PRODUCT-E --  *)
   | Product_vE_imp: forall e S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                           vq1 {HndpvQ1: NoDupElemvQ vq1} vq2 {HndpvQ2:
                           NoDupElemvQ vq2}
                            A1 {HndpAA1: NoDupElem A1} e1 A2 {HndpAA2: NoDupElem
                            A2} e2 ,
        vtypeImp e  S vq1 (A1, e1) →
         vtypeImp e  S vq2 (A2, e2) →
         vqtype_inter_vq (A1, e1) (A2, e2)  =vqtype= (nil, litB false) →
          vtypeImp e  S (prod_v vq1 vq2) (vqtype_union_vq (A1, e1) (A2, e2))
   (*  -- SETOP-E --  *)
   | SetOp_vE_imp: forall e S {HndpRS:NoDupRn (fst S)} {HndpAS: NODupElemRs S}
                         vq1 {HndpvQ1: NoDupElemvQ vq1} vq2 {HndpvQ2: NoDupElemvQ
                         vq2}
                          A1 {HndpAA1: NoDupElem A1} e1 A2 {HndpAA2: NoDupElem A2}
                          e2 op,
        vtypeImp e S vq1 (A1, e1) →
         vtypeImp e S vq2 (A2, e2) →
          equiv_vqtype (A1, e1) (A2, e2) →
           vtypeImp e S (setU_v op vq1 vq2) (A1, e1).

 Notation "{ e , S |- vq | vt }" := (vtypeImp e S vq vt) (e at level 200).
```

C.2 Correctness of Implicit VRA Type System

C.2.1 Correctness of Explicitly Annotating Function w.r.t. Implicit VRA type System

```
Lemma ImpQ_ImpType_ExpQ_ImpType e S q A:
   { e , S |-  q   | A }  →
   exists A', { e , S |- [q]S | A' } ∧  A  =vqtype= A'.
Proof. intro HImpQ.
(* From Lemma in Appendix C.2.3  *)
apply ImpQ_ImpType_implies_ExpQ_ImpType in HImpQ as HExpQ.
destruct HExpQ as [A' HExpQ].
(* From Lemma in Appendix  C.2.4  *)
apply (ImpQ_ImpType_Equiv_ExpQ_ImpType HImpQ) in HExpQ as HEquiv.
```

```
exists A'. eauto.
Qed.
```

C.2.2 Correctness of Implicit VRA Type System For Explicitly Annotated V-query

```
Lemma ExpQ_ImpType_ExpQ_ExpType e S q A (HndpQ: NoDupElemvQ q):
   { e , S |-  [q]S   | A }  →
   exists A', { e , S |= [q]S | A' } ∧  A  =vqtype= A'.
Proof. intro HImp.
(* From Lemma in Appendix  C.2.5  *)
apply ExpQ_ImpType_implies_ExpQ_ExpType in HImp as HExp;
try assumption.
destruct HExp as [A' HExp].
(* From Lemma in Appendix  C.2.6  *)
apply (ExpQ_ImpType_Equiv_ExpQ_ExpType HndpQ HImp) in HExp as HEquiv.
exists A'. eauto.
Qed.
```

C.2.3 ImpQuery ImpType implies ExpQuery ImpType

```
Lemma ImpQ_ImpType_implies_ExpQ_ImpType e S q A:
   { e , S |- q | A }  →
   exists A', { e , S |- [q]S | A' }.
 Proof.
 generalize dependent A.
 generalize dependent e.
 induction q; destruct A as (A, ea);
 intros HImp.

 { (* Relation - E *)

 destruct v as (rn, (A_, e_)).
 simpl in HImp. simpl.
 inversion HImp as [| eInv SInv HndpRSInv HndpASInv rnInv HeRInv A_Inv
                    A'Inv HndpA'Inv e_Inv e'Inv
                    HInVRInv | | | | |]; subst.

 rename e'Inv into e'.
 apply InVR_findVR in HInVRInv as HInFindInv; try assumption.

 rewrite HInFindInv.
```

```
unfold getvs, getf. simpl.

exists ((A, e ∧ (F) e')).

apply Relation_vE_imp; try assumption.

}

{ (* Projection - E *)
simpl in HImp. simpl.

destruct (vtypeImpNOTC (litB true) S ([q] S)) as (Aqs, eqs) eqn:HqST.

destruct a as (Ap, ep).

inversion HImp as [| |
                   eInv SInv HndpRSInv HndpASInv vqInv HndpvQInv
                   e'Inv A'Inv HndpAA'Inv QInv HndpQInv
                   HqInv HsbsmpInv | | | |]; subst.

apply IHq in HqInv as Hqs. destruct Hqs as [(Aqse, eqse) Hqs].
apply vtypeImpNOTC_correct in Hqs as HqSTine; try assumption.

apply NoDupElem_vtypeImpNOTC' in HqST as HndpelemAqs; try assumption.

apply eq_equiv_vqtype in HqST. (*as HqSTeqv.*)

apply (contex_intro_NOTC (litB true))
with (e':=e) (eq':= (eqs ∧ (F) e) ) in HqST; try assumption; try reflexivity.

assert(Htrue_e: (litB true ∧ (F) e) =e= e ).
{ unfold equivE. simpl. reflexivity. }

apply (contex_equiv_NOTC) with (S:=S) (q:=[q] S) in Htrue_e; try assumption.

rewrite HqST in Htrue_e.
rewrite HqSTine in Htrue_e.

exists (vqtype_inter_vq (vqtype_inter_vq (Ap, ep) (Aqs, eqs)) (Aqse, eqse)).
apply Project_vE_imp; try assumption.

all: apply NoDupElem_vtypeImp in Hqs as HndpAqse; try assumption;
apply NoDupElemvQ_ImptoExp with (S:=S) in HndpvQInv; try assumption;
auto.

{ unfold vqtype_inter_vq. simpl. simpl in *.
```

```
    apply NoDupElem_velems_inter; assumption. }

}

{ (* Selection - E *)
simpl in HImp. simpl.

destruct (vtypeImpNOTC (litB true) S ([q] S)) as (Aqs, eqs) eqn:HqST.

inversion HImp as [| | |
                   eInv SInv HndpRSInv HndpASInv vqInv HndpvQInv
                   A'Inv HndpAA'Inv e'Inv vcInv
                   HqInv HcondInv | | | ]; subst.

apply IHq in HqInv as Hqs. destruct Hqs as [(Aqse, eqse) Hqs].
apply vtypeImpNOTC_correct in Hqs as HqSTine; try assumption.

exists ((Aqse, eqse)).
apply Select_vE_imp; try assumption.

all: apply NoDupElem_vtypeImp in Hqs as HndpAqse; try assumption;
apply NoDupElemvQ_ImptoExp with (S:=S) in HndpvQInv; try assumption.

pose (ImpQ_ImpType_Equiv_ExpQ_ImpType HqInv Hqs) as HqeqvqS.

apply vcondtype_equiv with (e:=e) (vc:=v) in HqeqvqS; auto.
}

4:{ (* Empty - E *)
inversion HImp; subst.
simpl. exists (nil, litB false).
assumption.
}

all: (* Choice- E / Porduct - E / SetOP -E *)
inversion HImp as [| | |
                   |
                   eInv e'Inv SInv HndpRSInv HndpASInv
                   vq1Inv HndpvQ1Inv vq2Inv HndpvQ2Inv
                   A1Inv HndpAA1Inv e1Inv A2Inv HndpAA2Inv e2Inv
                   Hq1Inv Hq2Inv
                   |
                   eInv SInv HndpRSInv HndpASInv
                   vq1Inv HndpvQ1Inv vq2Inv HndpvQ2Inv
                   A1Inv HndpAA1Inv e1Inv A2Inv HndpAA2Inv e2Inv
                   Hq1Inv Hq2Inv HInterInv
                   |
                   eInv SInv HndpRSInv HndpASInv
                   vq1Inv HndpvQ1Inv vq2Inv HndpvQ2Inv
                   A1Inv HndpAA1Inv e1Inv A2Inv HndpAA2Inv e2Inv opInv
                   Hq1Inv Hq2Inv HEquivInv ]; subst;

apply IHq1 in Hq1Inv as Hq1S; apply IHq2 in Hq2Inv as Hq2S;
destruct Hq1S as [(A1, e1) Hq1S];
destruct Hq2S as [(A2, e2) Hq2S];
apply NoDupElem_vtypeImp in Hq1S as HndpA1; try assumption;
apply NoDupElem_vtypeImp in Hq2S as HndpA2; try assumption;
try (apply NoDupElemvQ_ImptoExp; assumption);
simpl;
 try( exists (vqtype_union_vq (A1, e1) (A2, e2));
      apply Choice_vE_imp with (A2:=A2) (e2:=e2)
    );
 try( exists (vqtype_union_vq (A1, e1) (A2, e2));
      apply Product_vE_imp with (A2:=A2) (e2:=e2)
    );
 try( exists (A1, e1);
      apply SetOp_vE_imp with (A2:=A2) (e2:=e2)
    );
 try assumption;
 try (apply NoDupElemvQ_ImptoExp; assumption);
 pose (ImpQ_ImpType_Equiv_ExpQ_ImpType Hq1Inv Hq1S) as Hq1eqvq1S;
 pose (ImpQ_ImpType_Equiv_ExpQ_ImpType Hq2Inv Hq2S) as Hq2eqvq2S.


{ (* Product_vE_imp → velems_inter_vq (A1, e1) (A2, e2)  =vqtype= [] *)
  pose (vqtype_inter_vq_equiv ) as HInterEqv.
  apply HInterEqv with (A:=(A1Inv, e1Inv)) (A':=(A1, e1)) in Hq2eqvq2S as
  HInterEqv';
  try (simpl; assumption).
  clear HInterEqv. rename HInterEqv' into HInterEqv.
  rewrite HInterInv in HInterEqv. symmetry. assumption.
}

{ (* SetOp_vE_imp → (A1, e1)  =vqtype= (A2, e2) *)
  symmetry in Hq1eqvq1S.
  transitivity (A, ea); try assumption.
  transitivity (A2Inv, e2Inv); try assumption.
}

Qed.
```

C.2.4 ImpQuery ImpType Equiv ExpQuery ImpType

```
Lemma ImpQ_ImpType_Equiv_ExpQ_ImpType e S q A A':
   { e , S |-  q   | A }  →
   { e , S |- [q]S | A' } →
    A  =vqtype= A'.

 Proof.
 generalize dependent A'. generalize dependent A. generalize dependent e.
 induction q; destruct A as (A, ea);  destruct A' as (A', ea'); intros HImp HExp.

 { (* Relation - E *)
 inversion HImp; subst. simpl ImptoExp in HExp.

 apply InVR_findVR in H3 as HInFind. rewrite HInFind in HExp.

 unfold getvs in HExp. unfold getf in HExp.

 simpl in HExp. inversion HExp; subst.

 apply InVR_findVR in H2 as HInFind'. rewrite HInFind in HInFind'.

 inversion HInFind'; subst. reflexivity. all: assumption.
 }

 { (* Projection - E *)

 simpl in HImp.
 simpl in HExp.

 destruct (vtypeImpNOTC (litB true) S ([q] S)) as (Aqs, eqs) eqn:HqST.


 destruct a as (Ap, ep).

 inversion HImp as [| |
                   eImp SImp HndpRSImp HndpASImp vqImp HndpvQImp
                   e'Imp A'Imp HndpAA'Imp QImp HndpQImp
                   HqImp HsbsmpImp | | | |]; subst.
 inversion HExp as [| |
                   eExp SExp HndpRSExp HndpASExp vqExp HndpvQExp
                   e'Exp A'Exp HndpAA'Exp QExp HndpQExp
                   HqExp HsbsmpExp| | | |]; subst.

 apply NoDupElem_vtypeImpNOTC' in HqST as HndpelemAqs; try assumption.
 apply eq_equiv_vqtype in HqST.

 apply (contex_intro_NOTC (litB true))
 with (e':=e) (eq':= (eqs ∧ (F) e) ) in HqST; try assumption; try reflexivity.

 apply vtypeImpNOTC_correct in HqExp as HqSTine; try assumption.

 apply IHq with (A:=(A'Imp, e'Imp)) in HqExp as Hqe; try assumption.

 apply eq_equiv_vqtype in HqSTine.

 (* equivalent context intro *)
 assert(Htrue_e: (litB true ∧ (F) e) =e= e ).
 { unfold equivE. simpl. reflexivity. }

 apply (contex_equiv_NOTC) with (S:=S) (q:=[q] S) in Htrue_e; try assumption.
 rewrite HqST in Htrue_e. rewrite HqSTine in Htrue_e.

 apply vqtype_inter_vq_equiv_Imp_Exp with (Ap:=Ap) (ep:=ep) (A'Imp:=A'Imp)
 (e'Imp:=e'Imp)
 in Htrue_e as Hvqtype_inter; try (simpl; assumption).
 }

 { (* Selection - E *)
 simpl in HImp.
 simpl in HExp.

 destruct (vtypeImpNOTC (litB true) S ([q] S)) as (Aqs, eqs) eqn:HqST.


 inversion HImp as [| | |
                   eImp SImp HndpRSImp HndpASImp vqImp HndpvQImp
                   A'Imp HndpAA'Imp e'Imp vcImp
                   HqImp HcondImp | | | ]; subst.
 inversion HExp as [| | |
                   eExp SExp HndpRSExp HndpASExp vqExp HndpvQExp
                   A'Exp HndpAA'Exp e'Exp vcExp
                   HqExp HcondExp | | |]; subst.

 apply NoDupElem_vtypeImpNOTC' in HqST as HndpelemAqs; try assumption.

 apply IHq with (A':=(A', ea')) in HqImp; assumption.
 }

 4:{ (* Empty - E *)
     inversion HImp; subst. simpl ImptoExp in HExp.
     inversion HExp; subst. reflexivity.
```

```
  }

 all: inversion HImp as
 [ |
 | | | e0 f0 S0 HnS HnAS
       q10 HnQ1 q20 HnQ2
       A1 HnA1 e1 A2 HnA2 e2
       Hq1 Hq2
     | e0 S0 HnS HnAS
       q10 HnQ1 q20 HnQ2
       A1 HnA1 e1 A2 HnA2 e2
       Hq1 Hq2
     | e0 S0 HnS0 HnAS0
       q10 HnQ10 q20 HnQ20
       A1 HnA1 e1 A2 HnA2 e2 op
       Hq1 Hq2 HEquiv]; subst;
 simpl in HExp; inversion HExp as
 [ |
 | | | e0 f0 S0 HnSs HnASs
       q10 HnQ1s q20 HnQ2s
       A1s HnA1s e1s A2s HnA2s e2s
       Hq1s Hq2s
     | e0 S0 HnSs HnASs
       q10 HnQ1s q20 HnQ2s
       A1s HnA1s e1s A2s HnA2s e2s
       Hq1s Hq2s
     | e0 S0 HnSs HnASs
       q10 HnQ1s q20 HnQ2s
       A1s HnA1s e1s A2s HnA2s e2s ops
       Hq1s Hq2s HEquivs]; subst.

 1, 2: apply IHq1 with (A':=(A1s, e1s)) in Hq1;
 apply IHq2 with (A':=(A2s, e2s)) in Hq2;
 try (assumption);
 apply vqtype_union_vq_equiv with (A:=(A1, e1)) (A':=(A1s, e1s)) in Hq2;
 assumption.

 apply IHq1 with (A':=(A', ea')) in Hq1; assumption.

 Qed.
```

C.2.5 ExpQuery ImpType implies ExpQuery ExpType

```
Lemma ExpQ_ImpType_implies_ExpQ_ExpType e S q A (HndpQ: NoDupElemvQ q):
  { e , S |- [q]S | A } →
  exists A', { e , S |= [q]S | A' }.
Proof.
generalize dependent A.
generalize dependent e.
induction q; destruct A as (A, ea);
intros HImp.

{ (* Relation - E *)
destruct v as (rn, (A_, e_)).
simpl in HImp.

destruct (findVR rn S) as (rn_, (Ar, er)) eqn: HfindVR.

unfold getvs, getf in HImp. simpl in HImp.


inversion HImp as [| eImp' SImp' HndpRSImp' HndpASImp'
                   rnImp' HeRImp' A_Imp'  A'Imp' HndpA'Imp' e_Imp' e'Imp'
                   HInVRImp |
                   | | | |]; subst.

apply InVR_findVR in HInVRImp
as HInFindImp; try assumption.

rewrite HfindVR in HInFindImp.
inversion HInFindImp; subst.

simpl. rewrite HfindVR.
unfold getvs, getf. simpl.


exists (A, (e ∧ (F) e'Imp')).
apply Relation_vE; try assumption.

}

{ (* Projection - E *)
rename a into Q.
(*

HImp: {e, S |- [proj_v Q q] S | (A, ea)}
--------------------------------------------------
exists A' : vqtype, {e, S |= [proj_v Q q] S | A'}

Proof sketch:

HImp: {e, S |- [proj_v Q q] S | (A, ea)}
```

```
S1. simpl ([] S) (in HImp and Goal) with

1. vtypeImpNOTC (litB true) S ([q] S) := (Aqs, eqs) -- HqST
:= { litB true, S |- ([q] S) | (Aqs, eqs) }
2. Q/-\Qs = (vqtype_inter_vq Q (Aqs, eqs))

HImp: {e, S |- proj_v (Q/-\Qs) ([q] S) | (A, ea)}
--------------------------------------------------
exists A' : vqtype, {e, S |= proj_v (Q/-\Qs) [q] S | A'}
*)

simpl in HImp. simpl.
destruct (vtypeImpNOTC (litB true) S ([q] S)) as (Aqs, eqs) eqn:HqST.

remember (vqtype_inter_vq Q (Aqs, eqs)) as QiQs.

(*
S2. inversion HImp to get (A, ea)
3. {e, S |- ([q] S) | (Aqse, eqse)} - HqImp
4. Q/-\Qs/-\Qse := vqtype_inter_vq (P/-\Qt) (Aqse, eqse)

HImp: {e, S |- proj_v (Q/-\Qs) ([q] S) | Q/-\Qs/-\Qse }
*)
inversion HImp as [| |
                   eImp SImp HndpRSImp HndpASImp vqImp HndpvQImp
                   e'Imp A'Imp HndpAA'Imp QImp HndpQImp
                   HqImp HsbsmpImp | | | |]; subst.

(*S1.1 *) apply NoDupElem_vtypeImpNOTC' in HqST as HndpelemAqs; try
assumption.
rename e'Imp into eqse.
rename A'Imp into Aqse.
remember (vqtype_inter_vq Q (Aqs, eqs)) as QiQs.
remember (velems_inter (fst QiQs) Aqse) as QiQsiQseA.
remember (snd QiQs ∧ (F) eqse) as QiQsiQsee.

(*
S3. relate 1-HqST 3-HqImp with context intro (litB true → litB true ∧ e → e *
3Hqst:{ litB true,      S |- ([q] S) | (Aqs, eqs   ) } →
S3.1  { litB true ∧  e, S |- ([q] S) | (Aqs, eqs∧ e) } →
S3.2  {              e, S |- ([q] S) | (Aqs, eqs∧ e) } →

S3.3 from 3:{ e, S |- ([q] S) | (Aqse, eqse) } and S3.2
4.1: HqsA: Aqse  =vqtype= Aqs
4.2: Hqse: eqse =e= eqs ∧  e
3: { e, S |- ([q] S) | (Aqs, eqse) } -- HqImp *)

(*S3.1 intro e in context: litB true → litB true ∧ e *)
apply eq_equiv_vqtype in HqST.

apply (contex_intro_NOTC (litB true))
with (e':=e) (eq':= (eqs ∧ (F) e) ) in HqST; try assumption; try reflexivity.

(*S3.2*)
(* litB true ∧ e =e= e *) assert(HqsAe: (litB true ∧ (F) e) =e= e ).
{ unfold equivE. simpl. reflexivity. }

(* contex equiv implies type euiv → *)
apply (contex_equiv_NOTC) with (S:=S) (q:=[q] S) in HqsAe; try assumption.

(* inductive type to type function - ([q] S) in e *)
apply vtypeImpNOTC_correct in HqImp as HqImpTine; try assumption.

rewrite HqST in HqsAe.
rewrite HqImpTine in HqsAe.

(*
S4. get exp type from IHq that is equiv to imp

S4.1 apply IHq in 4 to get 5
Hexp: { e, S |= ([q] S) | (Aqse', eqse') } ---- HqExp

S4.2 apply imp exp type quiv to 4 and 5
HqsAe': (Aqse' =vset= Aqse) ∧ (eqse' =e= eqs ∧ e)
*)

apply IHq in HqImp as HqExp.
destruct HqExp as [(Aqse', eqse') HqExp].
apply NoDupElem_vtype in HqExp as HndpAqse'; try assumption.

(*S4.2 ExpQ_ImpType_Equiv_ExpQ_ExpType *)
pose ExpQ_ImpType_Equiv_ExpQ_ExpType as HqsAe'.
apply HqsAe' with (A:=(Aqse, eqse)) in HqExp as HqsAe''; try assumption.
clear HqsAe'. rename HqsAe'' into HqsAe'.

(*
S5. exists (Q/-\Qs)^^e (in Goal)

-----------------------------------------------------
{e, S |= proj_v (Q/-\Qs) [q] S | (Q/-\Qs)^^e} *)

exists (QiQs^^e).
```

```
(*
S6. apply Proj_v in Goal with (A' := Aqse') ∧ (e' := eqse')
--------------------------------------------(1/2)
{ e, S |= ([q] S) | (Aqse', eqse') }

S7. assumption 7. Qed.

--------------------------------------------(2/2)
subset_vqtype (Q/-\Qs)^^e (Aqse', eqse')

*)

apply Project_vE with (A':=Aqse') (e':=eqse');(*S7*)try assumption.


(*
S8. (Q/-\Qs)^^e → (Q/-\(Aqs, eqs))^^e → (Q/-\(Aqs, eqs∧ e))

S9. Aqse' =vset= Aqs ; eqsq' =e= eqse =e= eqs ∧ e

-----------------------------------------------
subset_vqtype (Q/-\(Aqs, eqs∧ e)) (Aqs, eqs∧ e)

S10. subset_vqtype (A/-\B) B

Qed.

*)

rewrite HeqQiQs. destruct Q as (Aq, eq).
unfold addannot. simpl fst. simpl snd.
rewrite ← subset_vqtype_correctness;
try (simpl; assumption).
unfold subset_vqtype_exp, subset_velems_exp. intros.


destruct H as [HIn He]. apply In_config_true with (c:=c) in HIn; try assumption.
unfold avelems_velems in HIn. simpl fst in *. simpl snd in *.
rewrite In_config_exists_true. unfold avelems_velems. simpl fst. simpl snd.

rewrite configVElemSet_push_annot in *. Search velems_inter.

simpl in HIn.

simpl.
```

```
unfold " =vqtype=", "=avset=" in *. simpl in *. specialize HqsAe' with c.
specialize HqsAe with c.


destruct ((E[[ eq]] c) && (E[[ eqs]] c) && (E[[ e]] c)) eqn:Heqeqse.
{ rewrite <- In_config_exists_true in HIn. destruct HIn as [eInter HIn].
  apply In_velems_inter in HIn.
  rewrite In_config_exists_true in HIn.

  assert (Heqse: (E[[ eqs]] c) && (E[[ e]] c) = true).
  { rewrite <- andb_assoc in Heqeqse. rewrite andb_true_iff in Heqeqse.
    destruct Heqeqse; assumption. }

  rewrite Heqse in HqsAe.

  destruct (E[[ eqse']] c);

   rewrite HqsAe' in HqsAe;
   unfold "=set=" in HqsAe; specialize HqsAe with x;
   destruct HqsAe as [HqsAeIn HqsAeC];
   rewrite <- HqsAeIn; auto.
}
{ destruct HIn. }

all: rewrite HeqQiQs in HndpQImp; unfold vqtype_inter_vq in HndpQImp;
simpl in HndpQImp; try (simpl; assumption).

all: inversion HndpQ; subst; auto.

}

{ (* Selection - E *)
simpl in HImp. simpl.
inversion HndpQ as [ | | c' q' Hndpq | | | |]; subst.
inversion HImp as [| | |
                   eImp SImp HndpRSImp HndpASImp vqImp HndpvQImp
                   A'Imp HndpAA'Imp e'Imp vcImp
                   HqImp HcondImp | | |]; subst.

apply IHq in HqImp as HqExp; try auto.
destruct HqExp as [(A', ea') HqExp].

exists (A', ea'). apply Select_vE; try assumption.

apply NoDupElem_vtype in HqExp as HndpAqse'; try assumption.

(*S4.2 ExpQ_ImpType_Equiv_ExpQ_ExpType *)
```

```
pose (ExpQ_ImpType_Equiv_ExpQ_ExpType Hndpq HqImp HqExp) as Hqimpexp;
try assumption.

apply vcondtype_equiv with (e:=e) (vc:=v) in Hqimpexp; assumption.

}

4: { (* Empty - E *)
simpl in HImp. inversion HImp; subst.
simpl. exists (nil, litB false).
apply EmptyRelation_vE; try assumption. }

(* Choice - E/ Product - E/ SetOp - E *)
all: simpl in  HImp; simpl;
inversion HndpQ as [| |
                    | f' q1' q2' Hndpq1 Hndpq2
                    | q1' q2' Hndpq1 Hndpq2
                    | op' q1' q2' Hndpq1 Hndpq2 |]; subst;
inversion HImp as [| | |
                   |
                   eImp e'Imp SImp HndpRSImp HndpASImp
                   vq1Imp HndpvQ1Imp vq2Imp HndpvQ2Imp
                   A1Imp HndpAA1Imp e1Imp A2Imp HndpAA2Imp e2Imp
                   Hq1Imp Hq2Imp
                   |
                   eImp SImp HndpRSImp HndpASImp
                   vq1Imp HndpvQ1Imp vq2Imp HndpvQ2Imp
                   A1Imp HndpAA1Imp e1Imp A2Imp HndpAA2Imp e2Imp
                   Hq1Imp Hq2Imp HInterImp
                   |
                   eImp SImp HndpRSImp HndpASImp
                   vq1Imp HndpvQ1Imp vq2Imp HndpvQ2Imp
                   A1Imp HndpAA1Imp e1Imp A2Imp HndpAA2Imp e2Imp opImp
                   Hq1Imp Hq2Imp HEquivImp ]; subst;

apply IHq1 in Hq1Imp as Hq1Exp; try auto;
apply IHq2 in Hq2Imp as Hq2Exp; try auto;
(*1, 5, 9:  *)destruct Hq1Exp as [(A1Exp, e1Exp) Hq1Exp];
destruct Hq2Exp as [(A2Exp, e2Exp) Hq2Exp];
apply NoDupElem_vtype in Hq1Exp as HndpA1Exp; try assumption;
apply NoDupElem_vtype in Hq2Exp as HndpA2Exp; try assumption;
 try( exists (vqtype_union_vq (A1Exp, e1Exp) (A2Exp, e2Exp));
      apply Choice_vE with (A2:=A2Exp) (e2:=e2Exp)
    );
 try( exists (vqtype_union_vq (A1Exp, e1Exp) (A2Exp, e2Exp));
      apply Product_vE with (A2:=A2Exp) (e2:=e2Exp)
```

```
     );
  try( exists (A1Exp, e1Exp);
       apply SetOp_vE with (A2:=A2Exp) (e2:=e2Exp)
     );
 try assumption;
 pose (ExpQ_ImpType_Equiv_ExpQ_ExpType Hndpq1 Hq1Imp Hq1Exp) as Hq1impexp;
 pose (ExpQ_ImpType_Equiv_ExpQ_ExpType Hndpq2 Hq2Imp Hq2Exp) as Hq2impexp.

 { (* Product_vE_imp → velems_inter A1 A2 =vset= [] *)
   pose (vqtype_inter_vq_equiv ) as HInterEqv.
   apply HInterEqv with (A:=(A1Imp, e1Imp)) (A':=(A1Exp, e1Exp)) in Hq2impexp as
   HInterEqv';
   try (simpl; assumption).
   clear HInterEqv. rename HInterEqv' into HInterEqv.
   rewrite HInterImp in HInterEqv. symmetry. assumption.
 }

 { (* SetOp_vE_imp → (A1, e1)  =vqtype= (A2, e2) *)
   symmetry in Hq1impexp.
   transitivity (A, ea); try assumption.
   transitivity (A2Imp, e2Imp); try assumption.
 }

 Qed.
```

C.2.6 ExpQuery ImpType Equiv ExpQuery ExpType

```
Lemma ExpQ_ImpType_Equiv_ExpQ_ExpType e S q A A' (HndpQ: NoDupElemvQ q):
   { e , S |- [q]S | A }  →
   { e , S |= [q]S | A' } →
    A  =vqtype= A'.
 Proof.
 generalize dependent A'.
 generalize dependent A.
 generalize dependent e.
 induction q; destruct A as (A, ea);
 destruct A' as (A', ea');
 intros HImp HExp.

 { (* Relation - E *)
 destruct v as (rn, (A_, e_)).
 simpl in HImp.
 simpl in HExp.

 destruct (findVR rn S) as (rn_, (Ar, er)) eqn: HfindVR.
```

```
unfold getvs, getf in HImp. simpl in HImp.
unfold getvs, getf in HExp. simpl in HExp.

inversion HImp as [| eImp' SImp' HndpRSImp' HndpASImp'
                   rnImp' HeRImp' A_Imp' A'Imp' HndpA'Imp' e_Imp' e'Imp'
                   HInVRImp |
                   | | | |]; subst.

inversion HExp as [| eExp' SExp' HndpRSExp' HndpASExp'
                   rnExp' HeRExp' A'Exp' HndpA'Exp' e'Exp'
                   HInVRExp HsatExp |
                   | | | |]; subst.

apply InVR_findVR in HInVRImp
as HInFindImp; try assumption.

apply InVR_findVR in HInVRExp
as HInFindImp'; try assumption.

rewrite HInFindImp in HInFindImp'.
inversion HInFindImp'; subst.

reflexivity.

(* =vset= *)reflexivity.

(* =e= *)simpl_equivE. destruct (E[[ ea']] c) eqn:Hea.
apply HsatExp in Hea. simpl in Hea. rewrite Hea. eauto.
eauto. *)
}

{ (* Projection - E *)
simpl in HImp.
simpl in HExp.

destruct (vtypeImpNOTC (litB true) S ([q] S)) as (Aqs, eqs) eqn:HqST.

destruct a as (Ap, ep).

inversion HImp as [| |
                   eImp SImp HndpRSImp HndpASImp vqImp HndpvQImp
                   e'Imp A'Imp HndpAA'Imp QImp HndpQImp
                   HqImp HsbsmpImp | | | |]; subst.
inversion HExp as [| |
                   eExp SExp HndpRSExp HndpASExp vqExp HndpvQExp
                   e'Exp A'Exp HndpAA'Exp QExp HndpQExp
                   HqExp HsbsmpExp| | | |]; subst.

apply NoDupElem_vtypeImpNOTC' in HqST as HndpelemAqs; try assumption.

inversion HndpQ as [| | Q' q' HndpAp HndpvQq | | | |]; subst.

apply eq_equiv_vqtype in HqST.

apply (contex_intro_NOTC (litB true))
with (e':=e) (eq':= (eqs ∧ (F) e) ) in HqST; try assumption; try reflexivity.

apply vtypeImpNOTC_correct in HqImp as HqSTine; try assumption.

apply eq_equiv_vqtype in HqSTine.

(* equivalent context intro *)
assert(Htrue_e: (litB true ∧ (F) e) =e= e ).
{ unfold equivE. simpl. reflexivity. }

apply (contex_equiv_NOTC) with (S:=S) (q:=[q] S) in Htrue_e; try assumption.
rewrite HqST in Htrue_e. rewrite HqSTine in Htrue_e.

symmetry. rewrite vqtype_fexp_assoc.

apply vqtype_inter_vq_equiv_Imp_Exp with (Ap:=Ap) (ep:=ep) (A'Imp:=A'Imp)
(e'Imp:=e'Imp)
in Htrue_e as Hvqtype_inter; try reflexivity; try (simpl; assumption).

rewrite vqtype_inter_vq_equiv with (A':=(Ap, ep)) (B':=(Aqs, eqs ∧ (F) e)) in
Hvqtype_inter;
try auto; try (symmetry; assumption); try reflexivity.

}

{ (* Selection - E *)
simpl in HImp.
simpl in HExp.

destruct (vtypeImpNOTC (litB true) S ([q] S)) as (Aqs, eqs) eqn:HqST.


inversion HImp as [| | |
                   eImp SImp HndpRSImp HndpASImp vqImp HndpvQImp
                   A'Imp HndpAA'Imp e'Imp vcImp
                   HqImp HcondImp | | |]; subst.
inversion HExp as [| | |
                   eExp SExp HndpRSExp HndpASExp vqExp HndpvQExp
                   A'Exp HndpAA'Exp e'Exp vcExp
                   HqExp HcondExp | | |]; subst.

apply NoDupElem_vtypeImpNOTC' in HqST as HndpelemAqs; try assumption.

inversion HndpQ; subst.

apply IHq with (A':=(A', ea')) in HqImp; try assumption.
}

4:{ (* Empty - E *)
inversion HImp; subst. simpl ImptoExp in HExp.
inversion HExp; subst. reflexivity.
}

all: (* Choice - E / Product - E/ SetOp -E *)

simpl in HImp;
simpl in HExp;

inversion HndpQ as [| |
                    | f' q1' q2' Hndpq1 Hndpq2
                    | q1' q2' Hndpq1 Hndpq2
                    | op' q1' q2' Hndpq1 Hndpq2 | ]; subst;

inversion HImp as [| | |
                   |
                   eImp e'Imp SImp HndpRSImp HndpASImp
                   vq1Imp HndpvQ1Imp vq2Imp HndpvQ2Imp
                   A1Imp HndpAA1Imp e1Imp A2Imp HndpAA2Imp e2Imp
                   Hq1Imp Hq2Imp
                   |
                   eImp SImp HndpRSImp HndpASImp
                   vq1Imp HndpvQ1Imp vq2Imp HndpvQ2Imp
                   A1Imp HndpAA1Imp e1Imp A2Imp HndpAA2Imp e2Imp
                   Hq1Imp Hq2Imp HInterImp
                   |
                   eImp SImp HndpRSImp HndpASImp
                   vq1Imp HndpvQ1Imp vq2Imp HndpvQ2Imp
                   A1Imp HndpAA1Imp e1Imp A2Imp HndpAA2Imp e2Imp opImp
                   Hq1Imp Hq2Imp HEquivImp ]; subst;
inversion HExp as [| | |
                   |
                   eExp e'Exp SExp HndpRSExp HndpASExp
                   vq1Exp HndpvQ1Exp vq2Exp HndpvQ2Exp
                   A1Exp HndpAA1Exp e1Exp A2Exp HndpAA2Exp e2Exp
                   Hq1Exp Hq2Exp
```

```
                    |
                    eExp SExp HndpRSExp HndpASExp
                    vq1Exp HndpvQ1Exp vq2Exp HndpvQ2Exp
                    A1Exp HndpAA1Exp e1Exp A2Exp HndpAA2Exp e2Exp
                    Hq1Exp Hq2Exp HInterExp
                    |
                    eExp SExp HndpRSExp HndpASExp
                    vq1Exp HndpvQ1Exp vq2Exp HndpvQ2Exp
                    A1Exp HndpAA1Exp e1Exp A2Exp HndpAA2Exp e2Exp opExp
                    Hq1Exp Hq2Exp HEquivExp ]; subst;

apply (IHq1 Hndpq1 _ _ _ Hq1Imp) in Hq1Exp as Hq1Eq;
apply (IHq2 Hndpq2 _ _ _ Hq2Imp) in Hq2Exp as Hq2Eq;

(* 3: SetOp - E *) try assumption;

try ( apply vqtype_union_vq_equiv with (A:=(A1Imp, e1Imp)) (A':=(A1Exp, e1Exp)) in
Hq2Eq;
assumption).
Qed.
```