A Framework for Synthesizing Transactional Database Implementations in a Proof Assistant

by

Sorawit Suriyakarn

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

© 2017 Massachusetts Institute of Technology. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2017

Certified by . . . .
Adam Chlipala
Associate Professor without Tenure of Computer Science, MIT EECS
Thesis Supervisor

Accepted by . . . .
Christopher Terman
Chairman, Masters of Engineering Thesis Committee

A Framework for Synthesizing Transactional Database Implementations in a Proof Assistant

by

Sorawit Suriyakarn

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We propose CoqSQL, a framework for optimizing relational queries and automatically synthesizing relational database implementations in the Coq proof assistant, based on Anders Kaseorg's and Mohsen Lesani's Transactions framework. The synthesized codes support concurrent transaction execution on multiple processors and are accompanied with proofs certifying their correctness. The contributions include: (1) a complete specification of a subset of SQL queries and database relations, including support for indexes; and (2) an extensible, automated, and complete synthesis process from standard SQL-like specifications to executable concurrent programs.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor without Tenure of Computer Science, MIT EECS

Acknowledgments

First and foremost, I would like to give special thanks to Prof. Adam Chlipala for his support and guidance throughout the thesis project and my life at MIT. I would like to thank Anders Kaseorg and Mohsen Lesani for their incredible work on the Transactions project. I would also like to thank my UROP and SuperUROP supervisor Dr. Benjamin Delaware for his supervision of my earlier Coq projects.

Contents

1 Introduction **11**

1.1 Motivation 11

1.2 User Interaction 12

1.3 Workflow 14

2 Transactions Framework **17**

2.1 Terminology 17

2.2 Composable Components 20

2.3 Transactional Specifications 22

2.4 Predicated Structures 24

3 Data Representations **27**

3.1 Denotation 27

3.2 Type Universes 27

3.3 Trees, Schemas, and Tuples 29

3.4 Paths and Choices 31

3.5 Database Structures 35

4 SQL Specifications **37**

4.1 Database Abstract Representation 37

4.2 Helper Functions 38

4.3 Query Plans 39

4.4 SQL Methods 44

4.5 SQL Specification Objects 45

5 Implementations **47**

5.1 SQL Table Specification Objects 47

5.2 SQL Table Implementations 48

5.3 Naive Database Engine 50

5.4 Synthesis Module 51

6 Improving CoqSQL's Usability **55**

6.1 Named-to-Unnamed Translation 55

6.2 Simple SQL Type Universe 58

6.3 Evaluation 60

7 Related Work and Future Research **63**

7.1 Related Work 63

7.2 Future Research 64

List of Figures

1-1 Summary of CoqSQL's workflow 16

3-1 An example of a schema in CoqSQL 31
3-2 An example of a path structure in CoqSQL 33
3-3 An example of a choice structure in CoqSQL 34

4-1 Examples of translation from relational algebra terms to query plans 39

Chapter 1

Introduction

1.1 Motivation

Database management systems (DBMSes) are an important part of computer systems. From online shoppings to social networks, applications rely on databases to store billions of rows of information. Because of the scale, databases must be performant, reliable, and safe for concurrent usage. To standardize communication with DBMSes, the SQL query language has been developed. Developers often use SQL together with popular database systems, such as MySQL or PostgresSQL, to build application backends, believing that they are both performant and safe. However, despite being actively developed, those database systems are far from perfect. For instance, MySQL reports that there are more than 80,000 total bugs, and 10,000 of them are currently unresolved [3]. Similarly, PostgreSQL's bug report mailing list shows a large number of issue reports every month since its release [6].

On the other side of computer science research, formal methods have increasingly gained popularity over the past decades. Interactive theorem provers allow developers to write program specifications together with implementations, and prove their correctness. As a result, the developers can be sure that programs neither perform incorrectly nor contain undesired side effects. Among those, Coq [9] is a widely adopted system with its very expressive type systems and proof automation. In this project, we develop

a system in Coq that synthesizes relational database implementations from declarative specifications of SQL queries, with strong guarantees that synthesized databases are completely bug-free. The framework, called CoqSQL, provides a complete specification of an SQL subset, allowing users to write high-level programs on top of those SQL queries. The synthesis process is done automatically, with proof that the generated implementations are concurrency-safe and sound. In addition, CoqSQL is extensible by custom implementation strategies without sacrificing its core correctness.

1.2 User Interaction

We start by showing the usage of CoqSQL. Consider an example case of a bookstore database. A developer wants to build a database that supports storing information on user accounts, books, and book orders. To start, the database specification module must be declared. Note that database schemas and operations must implement `EngineArgs` module type in order for CoqSQL to automatically synthesize the implementation. Chapter 5 discusses the module constraints in detail. The developer then declares a database schema. In this example case of bookstore, the database contains three tables. CoqSQL's internal database structure can be constructed by compiling the high-level schema.

```
Module BookStoreDatabaseArgs <: EngineArgs.

  Definition schema ≔
    DATABASE ⟨
      TABLE "ACCOUNT" ⟨ "Id" ::: VARCHAR 20; "Balance" ::: UINT ⟩;
      TABLE "BOOK" ⟨ "Title" ::: VARCHAR 20; "Isbn" ::: VARCHAR 20; "Price" ::: UINT ⟩;
      TABLE "ORDER" ⟨ "AccountId" ::: VARCHAR 20; "BookIsbn" ::: VARCHAR 20 ⟩
    ⟩.

  Definition database ≔ COMPILE schema.
```

Next, the developer provides the set of SQL methods that may interact with the database in Gallina with augmented SQL-like style queries. In this case, there are four methods: `addAccount` creates a new account with a given ID and an initial balance;

`addBook` adds a book to the database; `placeOrder` places a book order and charges money from an account; and `removeAccount` deletes an account and its associated orders.

```
  Inductive method : Type → Type :=
  | addAccount : ⟦ VARCHAR 20 ⟧ → ⟦ UINT ⟧ → method unit
  | addBook : ⟦ VARCHAR 20 ⟧ → ⟦ VARCHAR 20 ⟧ → ⟦ UINT ⟧ → method unit
  | placeOrder : ⟦ VARCHAR 20 ⟧ → ⟦ VARCHAR 20 ⟧ → method 𝔹
  | removeAccount : ⟦ VARCHAR 20 ⟧ → method unit

  Definition implProg Ret (m : method Ret) : Program (SQLMethod database) Ret :=
    match m with
    | addAccount id bal ⇒
      _ ← INSERT INTO "ACCOUNT" VALUES (id ,, bal);
      Return tt
    | addBook isbn title price ⇒
      _ ← INSERT INTO "BOOK" VALUES (isbn ,, title ,, price);
      Return tt
    | placeOrder id isbn ⇒
      bal ← QUERY SELECT VAR("a.Balance") AS "bal"
                  FROM "ACCOUNT" AS "a" WHERE VAR("a.Id") == VAL(id);
      price ← QUERY SELECT VAR("b.Price") AS "price"
                    FROM "BOOK" AS "b" WHERE VAR("b.Isbn") == VAL(isbn);
      match bal, price with
      | [ bal' ], [ price' ] ⇒
        if price' ≤? bal' : 𝔹
        then _ ← INSERT INTO "ORDER" VALUES (id ,, isbn);
             _ ← UPDATE "ACCOUNT" SET "Balance" = VAR("Balance") - VAL(price')
                 WHERE VAR("Id") == VAL(id);
             Return true
        else Return false
      | _, _ ⇒ Return false
      end
    | removeAccount id ⇒
      _ ← DELETE FROM "ACCOUNT" WHERE VAR("Id") == VAL(id);
      _ ← DELETE FROM "ORDER" WHERE VAR("AccountId") == VAL(id);
      Return tt
    end.
End BookStoreDatabaseArgs.
```

The developer may now close the module and use the synthesis library to construct an efficient database implementation with proof of its correctness using one line of code! Every operation of the implementation is guaranteed to be both correct and atomic.

```
Module BookStoreDatabase := EngineDatabase BookStoreDatabaseArgs.

Check BookStoreDatabase.Impl.
(*
  BookStoreDatabase.Impl
    : Implementation (Abortable BookStoreDatabaseArgs.method)
*)

Check BookStoreDatabase.Impl_ok.
(*
  BookStoreDatabase.Impl_ok
    : Simulates
        (ThreadsBehavior BookStoreDatabase.Impl)
        (superposition
          (ThreadsBehavior
             (serialImplementation
                (abortableObject
                   (methodObject
                      (programObject (SQLObject BookStoreDatabaseArgs.database))
                      BookStoreDatabaseArgs.implProg)))))
        (pair nil BookStoreDatabase.T.tStructInitState)
        (eq (pair nil (SQLStateInit BookStoreDatabaseArgs.database)))
*)
```

`BookStoreDatabase.Impl` is the synthesized implementation, and `BookStoreDatabase.Impl_ok` is its proof of correctness (see Chapter 2 for explanation of the type signatures). Executable Haskell code can be extracted from the implementation. This concludes the most normal usage of CoqSQL.

1.3 Workflow

Before diving into the details, we summarize the step-by-step workflow of CoqSQL. The steps are as described below and in Figure 1-1.

1. A user declares the database schema and writes a high-level program specification of methods in a conventional SQL language.

2. CoqSQL compiles the high-level SQL methods into naive query plans. This translation process is unverified. Alternatively, the user may directly write query plans as specifications instead of using the conventional SQL language.

3. Optionally, query plans can get repeatedly optimized by user-provided optimization scripts. Each optimization step has a proof trace certifying its correctness.

4. CoqSQL synthesizes a database implementation based on the query plans. Naive implementations are provided in the framework, and the user is free to extend it with more implementation strategies and proofs of their correctness.

5. The implementation gets extracted to executable Haskell code.

The rest of the paper discusses the thesis project in detail. Chapter 2 provides an overview of the Transactions framework, which is the foundation of CoqSQL. Chapter 3 discusses Coq data structures used to build up complex specifications and implementations of databases. Chapter 4 describes the full formalization of unnamed SQL and relational algebra in CoqSQL. Chapter 5 walks through naive implementation strategies that CoqSQL provides, and possible ways to extend them. Chapter 6 describes an untrusted library in CoqSQL that enhances the user experience. Lastly, Chapter 7 discusses broadly about research progress in the joint database and formal method field, as well as possible ideas to improve the work.

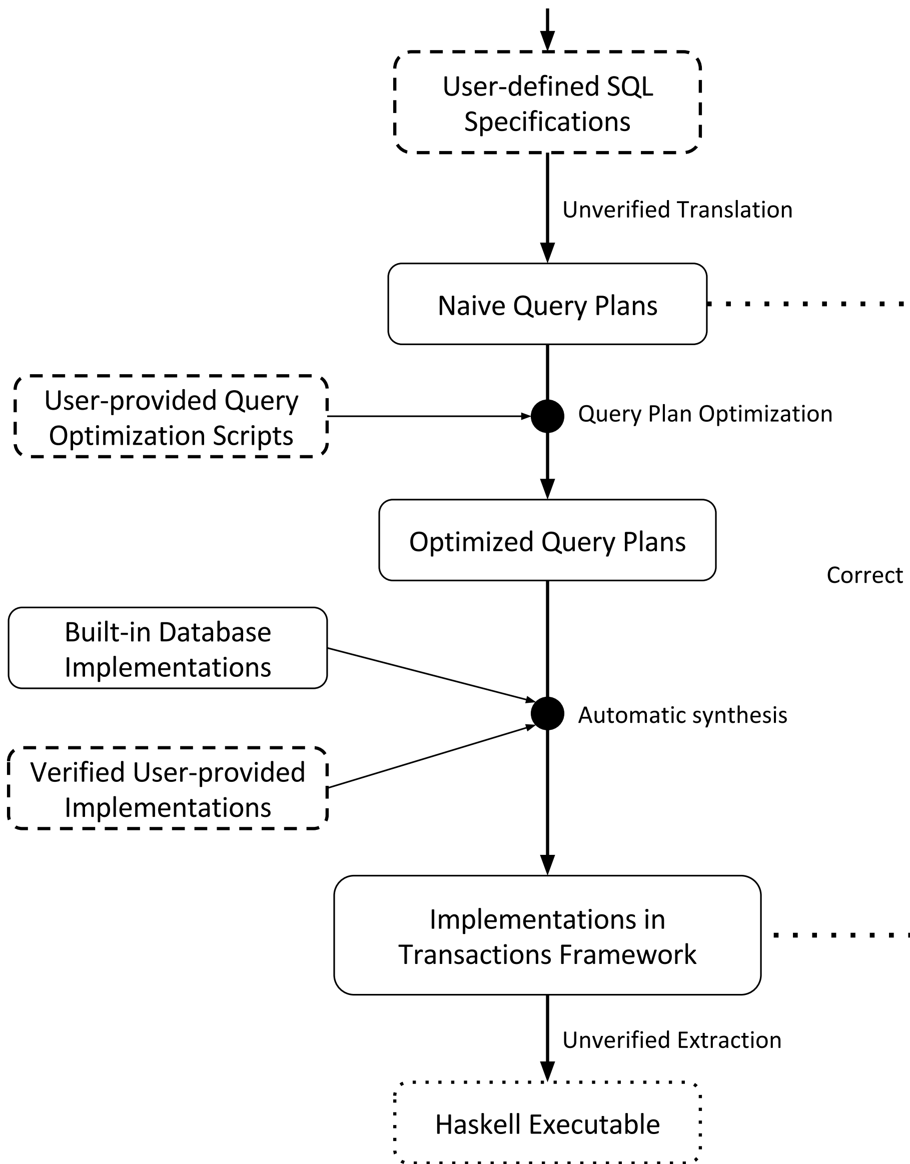

Figure 1-1: Summary of CoqSQL's workflow. Dashed boxes represent user-provided codes, solid boxes represent internal representations and implementations not exposed to users, and dotted boxes represent executable output of the synthesis process. Solid lines represent the main flow of the framework, thin lines represent interactions between the core framework and extra components, and dotted lines represent abstract relationships.

Chapter 2

Transactions Framework

CoqSQL is built on top of the Transactions framework by Anders Kaseorg and Mohsen Lesani, which is a system for constructing and verifying modular concurrent programs. It provides basic transactional data structures, notably concurrent map structures and registers, and transactional combinators that allow building more complex objects based on the simpler pieces. This chapter gives a summary of the framework.

2.1 Terminology

This section describes terminology that is used ubiquitously in the Transactions framework.

Method A *method type* is a set of methods of an interface parameterized by their return types. Arguments to any method are attached as part of the method name. For instance, the following is an interface for an integer register, consisting of three methods: `get` returns the current value of the register; `write` takes an integer, saves it to the register, and returns unit; and `cas` performs a compare-and-swap operation and returns whether it succeeds.

```
Inductive IntRegMethod : Type → Type :=
| get : IntRegMethod ℤ
| write : ℤ → IntRegMethod unit
| cas : ℤ → ℤ → IntRegMethod 𝔹.
```

Object An *object* is a transition system parameterized by a method type. It consists of an internal state type and a possibly nondeterministic transition relation. An object can be used to represent (1) a basic atomic instruction provided natively in a base machine, (2) a concrete implementation's behavior, or (3) an abstract specification of an interface.

```
Record Object (Method : Type → Type) :=
  {
    State : Type;
    EvalInstruction : ∀ Ret, Method Ret → State → Ret → State → Prop;
  }.
```

We continue the example by providing the proper semantics for each method. `IntRegObject` represents an interface of a base machine's instructions on an integer registers.

```
Inductive EvalIntRegMethod : ∀ Ret, IntRegMethod Ret → ℤ → Ret → ℤ → Prop :=
| evalGet : ∀ v, EvalIntRegMethod _ get v v v
| evalWrite : ∀ v v', EvalIntRegMethod _ (write v') v tt v'
| evalCasEq : ∀ v v', EvalIntRegMethod _ (cas v v') v true v'
| evalCasNeq : ∀ v v' v'', v ≠ v' → EvalIntRegMethod _ (cas v' v'') v false v.

Example IntRegObject : Object IntRegMethod :=
  {|
    State := ℤ;
    EvalInstruction := EvalIntRegMethod;
  |}.
```

Program A *program* is a shallowly embedded sequence of method calls. It is defined co-inductively to support infinite loops. The Transactions framework uses "`x ← m; p`" notation to abbreviate `(Bind m (fun x ⇒ p))`. The `Noop` constructor exists to allow

Coq's guardedness condition to be satisfied the case of an infinite loop that does not involve any method calls. This is necessary when implementing `flattenProgram` combinator (see Section 2.2).

```
CoInductive Program (Method : Type → Type) (Ret : Type) : Type :=
| Bind : ∀ Ret', Method Ret' → (Ret' → Program Method Ret) → Program Method Ret
| Return : Ret → Program Method Ret
| Noop : Program Method Ret → Program Method Ret.
```

Following is an example of an atomic increment program, which increases the value of the register by `x` and returns the updated value. It loops repeatedly until the compare-and-swap operation succeeds to ensure that the increment goes through.

```
CoFixpoint IntRegIncrement (x : ℤ) : Program IntRegMethod ℤ :=
  v ← get;
  b ← cas v (x + v);
  if b : 𝔹
  then Return (x + v)
  else IntRegIncrement x
```

Implementation An *implementation* of a method type describes a way to execute each method based on a lower level base interface. Extraction of an implementation gives an executable program that assumes base object methods. We finish the example of this section with an implementation of a simple counter interface.

```
Record Implementation (Method : Type → Type) :=
  {
    ImplInstruction : Type → Type;
    implBase : Object ImplInstruction;
    implProgram : ∀ Ret, Method Ret → Program ImplInstruction Ret;
  }

Inductive CounterInstruction : Type → Type :=
| add : ℤ → CounterInstruction ℤ
| reset : CounterInstruction unit.

Example counterImplProg Ret (m : CounterInstruction Ret)
  : Program IntRegMethod Ret :=
```

```
  match m with
  | add x ⇒ IntRegIncrement x
  | reset ⇒ _ ← write 0; Return tt
  end.

Example counterImpl : Implementation CounterInstruction :=
  {|
    ImplInstruction := IntRegMethod;
    implBase := IntRegObject;
    implProgram := counterImplProg;
  |}.
```

2.2 Composable Components

Complex structures can be built from simpler ones using combinators. The Transactions framework provides several combinators that allow composing methods, objects, programs, and implementations; and transforming one to another. With all the composable structures, we can build rich and complicated specifications and implementations in modular ways. At extraction time, composed implementations get automatically unfolded to sequences of native method calls, ensuring that no performance downside is introduced with this modular structure. CoqSQL relies on several composers in order to build rich database implementations. We shall see them in action in Chapter 5. Followings are some of the Transactions framework's combinators, with explanation.

Pair Two objects can be combined into a pair object. The method type is the disjoint union of the two method types, and the state type is the product of the two underlying states. Two implementations can also be combined into a pair implementation.

```
Definition PairMethod
  : (Type → Type) → (Type → Type) → Type → Type. □

Definition pairObject
  : ∀ InstructionL InstructionR : Type → Type,
      Object InstructionL → Object InstructionR →
      Object (PairInstruction InstructionL InstructionR). □
```

```
Definition pairImplementation
  : ∀ MethodL : Type → Type, Implementation MethodL →
      ∀ MethodR : Type → Type, Implementation MethodR →
        Implementation (PairMethod MethodL MethodR). □
```

Map Objects, programs, and implementations all support instruction mapping (i.e. method renaming). These combinators change all underlying method calls of a given structure based on a given mapping function.

```
Definition methodObject
  : ∀ Instruction : Type → Type,
      Object Instruction →
      ∀ Method : Type → Type,
        (∀ Ret : Type, Method Ret → Instruction Ret) → Object Method. □

Definition mapProgram
  : ∀ Instruction Instruction' : Type → Type,
      (∀ A : Type, Instruction A → Instruction' A) →
      ∀ Ret : Type, Program Instruction Ret → Program Instruction' Ret. □

Definition mapImplementation
  : ∀ Method Method' : Type → Type,
      (∀ Ret : Type, Method' Ret → Method Ret) →
        Implementation Method → Implementation Method'. □
```

Flatten Flatten combinators take a function that maps a high-level method call to a program with a low-level interface, and inline every high-level method call with the corresponding low-level program. It is important to note that without `Noop`, `flattenProgram` would not pass Coq's guardedness condition because replacing each method call in an infinite loop with a program consisting of only one `Return` results in a repeating loop of `flattenProgram` calls without any `CoFixpoint` constructor between each recursive call.

```
Definition flattenProgram
  : ∀ Method Instruction : Type → Type,
      (∀ A : Type, Method A → Program Instruction A) →
```

```
    ∀ Ret : Type, Program Method Ret → Program Instruction Ret. □

Definition flattenImplementation
  : ∀ Method : Type → Type, Implementation Method →
    ∀ Method' : Type → Type,
      (∀ Ret : Type, Method' Ret → Program Method Ret) → Implementation Method'. □
```

Abortable In a system with optimistic concurrency control, at any time, an operation may get aborted when a conflict arises. If it happens, the operation makes no change to the object's state and returns nothing. The Transactions framework includes combinators that facilitate building an abortable method or an abortable object.

```
Inductive Abortable (M : Type → Type) : Type → Type :=
| abortable : ∀ R : Type, M R → Abortable M (option R).

Definition abortableObject
  : ∀ Instruction : Type → Type,
    Object Instruction → Object (Abortable Instruction). □
```

2.3 Transactional Specifications

Objects in the Transactions framework can be used both as program behaviors and abstract specifications. This section describes the building blocks for constructing objects that represent sequential or concurrent semantics, and formalizes the core relationship between objects.

Sequential Semantics The Transactions framework provides an interface to convert between an implementation and an object. In the forward direction, an implementation can get *coarsened* to an object encoding its sequential semantics by abstracting it to a *program object*. A program object's transition relation is the reflexive transitive closure of its base object's transition relation. In other words, a program object accepts any transition that can be expressed as a sequence of the base object's transitions that finishes with a return clause.

```
Definition programObject (obj : Object Instruction) : Object (Program Instruction) :=
  {|
    State := obj.(State);
    EvalInstruction := EvalProgram obj;
  |}.

Definition coarsen {Method} (impl : Implementation Method) : Object Method :=
  methodObject (programObject impl.(implBase)) impl.(implProgram).
```

On the opposite direction, `serialImplementation` converts an object to its serial implementation, in which each method call is implemented as a simple program that calls the method and returns the result.

```
Definition serialImplementation (Method : Type → Type) (obj : Object Method) :=
  {|
    ImplInstruction := Method;
    implBase := obj;
    implProgram Ret m ⇒ x ← m; Return x
  |}
```

Concurrent Semantics The Transactions framework provides `ThreadsBehavior`, a function that converts an implementation to an object encoding its concurrent behavior. A thread object's state consists of a list of running programs plus an internal state, and its transition step involves nondeterministically picking threads to run and spawning new threads. For the purpose this thesis, we treat `ThreadsBehavior` as a black box and skip the detailed discussion about it.

```
Definition ThreadsBehavior
  : ∀ Method : Type → Type, Implementation Method → Object (Interaction Method). □
```

A *superposition* of an object captures all of the object's possible behaviors under nondeterminism. Its state is an ensemble of the base object's state, and its transition relation allows stepping from `sst` to `sst'` if `sst'` is not empty and only contains states that are reachable from at least one state in `sst`. Applying `superposition` and `ThreadsBehavior` on a sequential semantics object yields another object that represents the transactional semantics of the same program.

```
Definition superposition {Method} (obj : Object Method) : Object Method :=
  {|
    State := obj.(State) → Prop;
    EvalInstruction  A m sst a sst' :=
      (∃ y, sst' y) /\
      (∀ st',
          sst' st' → ∃ st,
            sst st /\ obj.(EvalInstruction) m st a st');
  |}.
```

Simulation Relation A *simulation relationship* between two objects is defined in the framework. `Simulates obj1 obj2 st1 s2` holds if every evaluation step of `obj1` from `s1` to `s1'` can be simulated via an evaluation step of `obj2` from `s2` to `s2'`, and simulation further relation holds true for `s1'` and `s2'`. As a result, every behavior trace produced by `obj1` is reproducible by at least one possible evaluation sequence of `obj2`. Thus, if `obj1` is a transition system of the synthesized implementation, `obj2` is an abstract desired property, and `Simulates obj1 obj2` holds for their initial states, then we are done proving correctness! CoqSQL uses the simulation relation as its main correctness lemma for the synthesized code (see Section 4.4).

```
CoInductive Simulates (Interaction : Type → Type)
                        (Behavior1 Behavior2 : Object Interaction)
                        (s1 : State Behavior1) (s2 : State Behavior2) : Prop :=
| SimStep :
  (∀ (R : Type) (int : Interaction R) (r : R) (s1' : State Behavior1),
    EvalInstruction Behavior1 int s1 r s1' →
      ∃ s2' : State Behavior2,
        EvalInstruction Behavior2 int s2 r s2' & Simulates Behavior1 Behavior2 s1' s2
    ')
  → Simulates Behavior1 Behavior2 s1 s2.
```

2.4 Predicated Structures

The Transactions framework uses transactional predication [1], a technique for building transactional implementations based on concurrent data structures. To promote modularity, the framework defines a *predicated structure* type, called `PStruct`, that

captures necessary properties required by transactional predication. Following is a part of the definition of `PStruct`, excluding properties that are only for internal use by the Transactions framework.

```
Record PStruct {Method Val} :=
  {
    ...
    ...
    pStructImplProg :
      ∀ R,
        Method R →
        Program (PairMethod LocatorMethod (BMapMethod Loc (option Val))) R;
    pStructSpec : Object Method;
    pStructSpecInit : pStructSpec.(State);
    ...
    ...
  }.
```

Basic predicated structures, such as atomic registers and concurrent hashmaps, are included in the framework. In addition, there exist various combinators for composing predicated structures, similar to what presented in 2.2.

```
Definition pairPStruct
  : ∀ (Method1 Method2 : Type → Type) (Val : Type),
      PStruct Method1 Val → PStruct Method2 Val →
      PStruct (Pair.PairMethod Method1 Method2) Val. □

Definition mapPStruct
  : ∀ (Method : Type → Type) (Val Val' : Type)
      (f : Val' → option Val) (g : Val → Val'),
        (∀ v : Val, f (g v) = Some v) →
        PStruct Method Val → PStruct Method Val'. □

Definition flattenPStruct
  : ∀ (Method Method' : Type → Type) (Val : Type),
      (∀ Ret : Type, Method' Ret → Program Method Ret) →
      PStruct Method Val → PStruct Method' Val. □
```

After constructing a predicated structure of a desired program, the developer builds a module satisfying `TStructPars` module type and feeds it to `TStructFun` module. This process produces an implementation of the program described in the given predicated

structure, and the initial state of its base object, together with the linearizability proof. The relevant part of `TStructPars` and `TStructFun` is presented below.

```
Module Type TStructPars.
  Parameter Method : Type → Type.
  Parameter Val : Type.
  Parameter pStruct : PStruct Method Val.
  ...
  ...
End TStructPars.

Module TStructFun (TStructArgs : TStructPars).
  ...
  ...
  Definition tStructImpl : Implementation (Abortable (Program TStructArgs.Method)).
  Definition tStructInitState : State (implBase tStructImpl).
  Definition tStructSpec : Object (Abortable (Program TStructArgs.Method)).
  ...
  Parameter tStructImplLin :
    Simulates
      (ThreadsBehavior tStructImpl)
      (superposition (ThreadsBehavior (serialImplementation tStructSpec)))
      (nil, tStructInitState)
      (eq (nil, pStructSpecInit TStructArgs.pStruct)).
  ...
  ...
End TStructFun.
```

Chapter 3

Data Representations

In this chapter, we discuss the data structures that are used to formalize CoqSQL's SQL specifications.

3.1 Denotation

An instance of `Denotation A B` represents a way to translate any object of type `A` to another object of type `B`. Once defined, the notation `⟦ a ⟧` resolves to the denotation function call on `a : A`.

```
Class Denotation (A B : Type) :=
  { denote : A → B }.
Notation "⟦ X ⟧" := (denote X).
```

Declaring this typeclass simplifies the process of writing specifications and proofs. We shall see its advantages more clearly in later sections.

3.2 Type Universes

A *type universe* is a collection of Coq types with (1) the proofs that every type has the properties of SQL types, (2) a set of operators between expressions, and (3) the universe's "unit" type.

The Properties of SQL Types

Decidability SQL datatypes must be decidable. In other words, there must be a computable procedure to determine if two objects of the same SQL type are equal. This property does not necessarily hold for arbitrary Coq types.

```
Class Decidable (τ : Type) ≔
  { eq_dec : ∀ p q : τ, {p = q} + {p ≠ q} }.
Notation "X =? Y" ≔ (if eq_dec X Y then true else false) (at level 20).
```

Finiteness A type `τ` is finite if there exists a duplicate-free list `all` that contains all elements of type `τ`. This list exists solely for proof purposes and is not a part of concrete implementations. The property is used to formalize SQL projections and the SQL EXISTS predicate (see Chapter 4 for details).

```
Class Finite (τ : Type) ≔
  {
    all : list τ;
    all_NoDup : NoDup all;
    all_ok : ∀ t : τ, In t all;
    fold ≔ fun (A : Type) (f : τ → A → A) (init : A) ⇒ fold_right f init all;
  }.
```

SQL Operators

The framework allows constant, unary, and binary operators on SQL expressions.

$$[\![\ c : \texttt{constant}\ \tau\ ]\!] : [\![\tau]\!]$$

$$[\![\ u : \texttt{unary}\ \tau\ \tau'\ ]\!] : [\![\tau]\!] \to [\![\tau']\!]$$

$$[\![\ b : \texttt{binary}\ \tau\ \tau'\ \tau''\ ]\!] : [\![\tau]\!] \to [\![\tau']\!] \to [\![\tau'']\!]$$

Every operator and its denotation must be declared during the initialization of a type universe. This allows CoqSQL to perform static check on queries and formulate

SQL relational semantics using only a type universe, since all operators are encoded there.

Unit Type

Additionally, any type universe must contain a unit type. A unit type is a very simple type with exactly one member. It is used as a placeholder for an empty tuple or an empty query context.

$$[\![\texttt{unit_t}]\!] = \{\texttt{tt_t}\}$$

Having formalized all the constraints, we now can define CoqSQL's type universe typeclass `Types`.

```
Class Types : Type :=
  {
    type : Type;
    denotationType : Denotation type Type;
    finiteType : ∀ τ : type, Finite ⟦τ⟧ ;
    decidableType : ∀ τ : type, Decidable ⟦τ⟧ ;
    unit_t : type;
    tt_t : ⟦unit_t⟧;
    unit_t_all : all (T := ⟦unit_t⟧) = tt_t :: nil;
    constant : type → Type;
    unary : type → type → Type;
    binary : type → type → type → Type;
    denotationConstant : ∀ τ, Denotation (constant τ) ⟦τ⟧;
    denotationUnary : ∀ τ τ', Denotation (unary τ τ') (⟦τ⟧ → ⟦τ'⟧);
    denotationBinary : ∀ τ τ' τ'', Denotation (binary τ τ τ') (⟦τ⟧ → ⟦τ'⟧ → ⟦τ'⟧);
  }.
```

3.3 Trees, Schemas, and Tuples

We proceed to describe the fundamental structures of objects in CoqSQL.

Trees In CoqSQL, complicated data structures are built by composing simpler ones, primarily using binary tree structure. CoqSQL has a reasonably standard

definition for generic binary trees.

```
Inductive Tree (A : Type) :=
| TreeNode (u v : Tree A)
| TreeLeaf (a : A).
Notation "[ X ]" := (TreeLeaf X).
Infix "⧫" := TreeNode (at level 20).
```

Schemas In the standard SQL definition, a schema is a collection of pairs of (x, τ), where x represents the name of an attribute, and τ represents the attribute's type. In CoqSQL, to avoid tedious naming resolution, we drop the attribute names completely and model a schema as a tree of types. Two symbols are used for schemas: σ refers to the schema of a concrete relation, and Γ refers to the schema of an SQL statement's typing context.

```
Definition Schema {TX : Types} := Tree type. (* 'type' resolves under 'TX' *)
```

Tuples A tuple is a dependent structure on a schema, represented as a nested pair of values corresponding to the schema's recursive structure. We use `⟨ σ ⟩` as the notation for the tuple type of schema `σ`.

```
Fixpoint Tuple {TX : Types} (σ : Schema) : Type :=
  match σ with
  | [ τ ] ⇒ ⟦ τ ⟧
  | σ_1 ⧫ σ_2 ⇒ Tuple σ_1 × Tuple σ_2
  end.
Notation "⟨ σ ⟩" := (Tuple σ).
```

As proven in the framework, it follows that any tuple in any type universe is both decidable and finite.

```
Instance decidableTuple {TX : Types} (heading : Schema)
  : Decidable (Tuple heading). □
Instance finiteTuple {TX : Types} (heading : Schema)
  : Finite (Tuple heading). □
```

As an example, consider a book schema σ_{book} with three attributes, and two tuples t_{CLRS} and t_{CoqArt} belonging to the schema.

$$\sigma_{book} = \{\text{Title} : \texttt{VARCHAR } 20,\ \text{Isbn} : \texttt{VARCHAR } 20,\ \text{Price} : \texttt{INTEGER}\}$$

$$t_{CLRS} = \{\text{Title} : \text{"Introduction to A..."},\ \text{Isbn} : \text{"9780262033848"},\ \text{Price} : 69\}$$

$$t_{CoqArt} = \{\text{Title} : \text{"Interactive Theor..."},\ \text{Isbn} : \text{"9783662079645"},\ \text{Price} : 79\}$$

Assuming that there exists a denotation instance for simple SQL types in context, we can write the schema and the tuples in CoqSQL as follows. Figure 3-1 shows the schema's visual representation.

```
Example BookSchema : Schema ≔ [VARCHAR 20] ⧫ ([VARCHAR 20] ⧫ [INTEGER]).
Example CLRSTuple : ⟨ BookSchema ⟩ ≔ ("9780262033848",("Introduction to A...",69)).
Example CoqArtTuple : ⟨ BookSchema ⟩ ≔ ("9783662079645",("Interactive Theor...",79)).
```

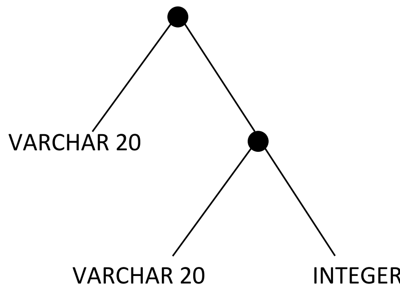

Figure 3-1: An example of a schema in CoqSQL. As shown, attribute names are not part of a schema.

3.4 Paths and Choices

We proceed to describe the data structures and the methods that can be used to manipulate composed structures.

Paths A path is a dependently typed object parameterized on a tree, representing a path from the root to one of the leaves. `Pick` is a procedure that takes a tree structure and a path, and returns the value at the corresponding leaf. Equality between two paths is decidable.

```
Inductive Path {A : Type} : Tree A → Type :=
| PHere : ∀ {T}, Path [T]
| PLeft : ∀ {X Y}, Path X → Path (X ⧫ Y)
| PRight : ∀ {X Y}, Path Y → Path (X ⧫ Y).

Fixpoint Pick {A : Type} (tree : Tree A) {struct tree} : Path tree → A :=
  match tree with
  | [ a ] ⇒ fun p ⇒ a
  | u ⧫ v ⇒ fun p ⇒
              match p with
              | PHere ⇒ idProp
              | PLeft p' ⇒ fun f g ⇒ f p'
              | PRight p' ⇒ fun f g ⇒ g p'
              end (Pick u) (Pick v)
  end.
Arguments Pick {_ _} _.

Instance decidablePath (A : Type) (tree : Tree A) : Decidable (Path tree). □
```

Following the previous example, we can define a path of `BookSchema` to point to one of its attributes. In this case, `IsbnPath` points to the schema's Isbn attribute. Figure 3-2 gives a visual representation of the path and the result of applying `Pick` on it.

```
Example IsbnPath : Path BookSchema := PRight (PLeft PHere).

Eval simpl in (Pick IsbnPath).
(*  = VARCHAR 20
    : type
*)
```

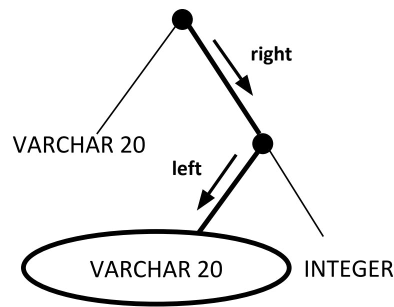

Figure 3-2: An example of a path structure in CoqSQL. As shown, `IsbnPath` points to a leaf of the tree, and picking it returns the value in such leaf.

Choices A choice is a dependently typed object parameterized on a schema and the type of the attribute to which the choice points. Similar to a path, a choice guarantees to point to one leaf. `Choose` is a procedure to project a value of a tuple given a choice. `Update` is a procedure to build a new tuple by changing a given tuple's field. Equality between two choices is decidable.

```
Inductive Choice {TX : Types} : Schema → type → Type :=
| CHere : ∀ {τ}, Choice [τ] τ
| CLeft : ∀ {X Y τ}, Choice X τ → Choice (X ⧫ Y) τ
| CRight : ∀ {X Y τ}, Choice Y τ → Choice (X ⧫ Y) τ.

Fixpoint Choose {TX : Types} {σ : Schema} {τ : type} (ch : Choice σ τ)
  : ⟨ σ ⟩ → ⟦ τ ⟧ :=
  match ch with
  | CHere ⇒ fun g ⇒ g
  | CLeft ch' ⇒ fun g ⇒ Choose ch' (fst g)
  | CRight ch' ⇒ fun g ⇒ Choose ch' (snd g)
  end.

Fixpoint Update {TX : Types} {σ : Schema} {τ : type} (ch : Choice σ τ)
  : ⟦ τ ⟧ → ⟨ σ ⟩ → ⟨ σ ⟩ :=
  match ch with
  | CHere ⇒ fun v _ ⇒ v
  | CLeft ch' ⇒ fun v g ⇒ (Update ch' v (fst g), snd g)
  | CRight ch' ⇒ fun v g ⇒ (fst g, Update ch' v (snd g))
  end.

Instance decidableChoice {TX : Types} (σ : Schema) (τ : type)
```

```
: Decidable (Choice σ τ). □
```

As an example, `IsbnChoice` is a choice of `BookSchema` that points to a leaf node holding value `VARCHAR 20`. Choosing it on a `CLRSTuple` projects CLRS's Isbn attribute. We can also build an almost similar tuple by updating `CLRSTuple`'s Isbn attribute with a new string. Figure 3-3 presents a visual representation of the operations.

```
Example IsbnChoice : Choice BookSchema (VARCHAR 20) := CRight (CLeft Here).

Eval simpl in (Choose IsbnChoice CLRSTuple)
(* = "9780262033848"
   : ⟦VARCHAR 20⟧
*)

Eval simpl in (Update IsbnChoice "N/A" CLRSTuple)
(* = ("Introduction to A...", ("N/A", 69))
   : ⟨ BookSchema ⟩
*)
```

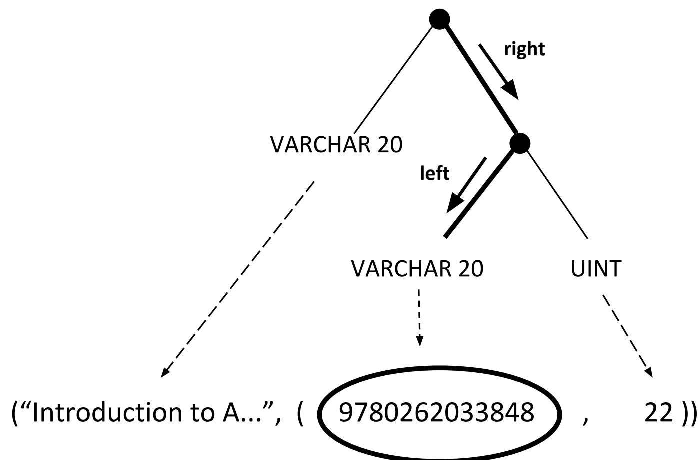

Figure 3-3: An example of a choice structure in CoqSQL. As shown, `IsbnChoice` points to a leaf whose value is similar to the choice's parameterized type.

3.5 Database Structures

With all the basic structures, we are ready to formulate SQL data structures. This section gives the definitions of the building blocks that form CoqSQL's database representation. Every definition is parameterized by a type universe.

Fields A *field* of a schema is a record consisting of the field's type and the choice that points to the field in the schema. There exist coercions from a field to its type and to its choice structure.

```
Record Field {TX : Types} (hd : Schema) : Type :=
  {
    fieldType :> type;
    fieldChoice :> Choice hd fieldType;
  }.
Arguments fieldType {_ _} _.
Arguments fieldChoice {_ _} _.
```

Continuing the example, we can declare a field `field_Isbn` of `BookSchema` that encodes the Isbn attribute as follows. Fields for other attributes can also be defined in a similar manner.

```
Example IsbnField : Field BookSchema :=
  {|
    fieldType := VARCHAR 20;
    fieldChoice :=  IsbnChoice;
  |}.
Example TitleField : Field BookSchema. □
Example PriceField : Field BookSchema. □
```

Relations and Indexes A *relation* consists of a schema and a collection of indexes in the form of a binary tree. Each index is a list of the schema's fields, analogously similar to SQL's conventional multi-column indexes. An index of a relation is a path of the relation's tree of indexes. There is a coercion that implicitly projects a schema out of a relation.

```
Record Relation {TX : Types} :=
  {
    heading :> Schema;
    indexes : Tree (list (Field heading));
  }.

Definition Index {TX : Types} (R : Relation) := Path R.(indexes).
```

We now can declare a relation `BookRelation` for our book table. Let us assume there are two indexes in this relation, one solely based on the Isbn attribute and the other based on the Title and Price attributes. Other relations can also be similarly defined.

```
Example BookRelation : Relation :=
  {|
    heading := BookSchema;
    indexes := [ IsbnField :: nil ] ♦ [ TitleField :: PriceField :: nil ];
  |}.
Example AccountRelation : Relation. □
Example OrderRelation : Relation. □
```

Databases and Tables A *database* is a collection of relations. We use the binary tree structure to compose multiple relations into a database. We define a table as a path on a database's tree of relations.

```
Definition Database {TX : Types} := Tree Relation.

Definition Table {TX : Types} (DB : Database) := Path DB.
```

We finish the chapter by building a database structure corresponding to the example shown in Chapter 1.

```
Example BookStoreDatabase : Database
  := [AccountRelation] ♦ ([BookRelation] ♦ [OrderRelation]).
```

Chapter 4

SQL Specifications

CoqSQL models an SQL database as a transition system with an abstract state. The transition system, called the *SQL specification object*, encapsulates the type of abstract state along with the state-transition step relation for any SQL statement. An SQL statement is modeled as a method call to an SQL specification object. Different SQL method calls may be provably equal if they share the same denotation. This chapter describes the formalization of CoqSQL's SQL specification object.

4.1 Database Abstract Representation

CoqSQL's SQL semantics is based on K-Relations [2, 5]. We define the abstract representation of a relation as a function from a tuple in that relation to the current number of its appearances. The abstract representation of a database is a function from a table to the abstract representation of the table's relation. The initial state contains no tuples in any table, and is thus represented by the function that returns zero in all cases.

```
Definition SQLState {TX : Types} (DB : Database) : Type
  ≔ ∀ (T : Table DB) (R : Relation ≔ Pick T), ⟨ R ⟩ → ℕ.

Definition SQLStateInit {TX : Types} (DB : Database) : SQLState DB ≔ fun _ _ ⇒ 0.
```

4.2 Helper Functions

We first discuss helper functions that we declare to facilitate the formalization.

Bool-to-nat Conversion CoqSQL uses notation `‖ X ‖` to represent the conversion from a Boolean `X` to a natural number. This conversion allows Boolean values to be used in relational algebra.

```
Definition 𝔹_to_ℕ (b : 𝔹) : ℕ :=
  match b with
  | true ⇒ 1
  | false ⇒ 0
  end.
Notation "‖ X ‖" := (𝔹_to_ℕ X) (at level 0).
```

Dependent Override Dependent override builds a new function similar to a given function but with exactly one key-value mapping change. Different from the regular override, `overrideDep` works on functions with dependent types, where the return result's type depends on the input's type. Note that the procedure requires decidablity of the function's argument type.

```
Section OverrideDep.
  Context {Key : Type} {Val : Key → Type} (eq_dec : ∀ p q : Key, {p = q} + {p ≠ q}).
  Definition Func := ∀ k, Val k.

  Definition overrideDep
             (f : Func) (k : Key) (val : Val k) : Func :=
    fun k' ⇒
      match eq_dec k' k with
      | left e ⇒ eq_rect_r (fun k'' : Key ⇒ Val k'') val e
      | right _ ⇒ f k'
      end.

  Lemma overrideDep_new_val
    : ∀ (f : Func) (k : Key) (v : Val k), overrideDep f k v k = v. ■
  Lemma overrideDep_old_val
    : ∀ (f : Func) (k k' : Key) (v : Val k), k ≠ k' → overrideDep f k v k' = f k'. ■
End OverrideDep.
```

4.3 Query Plans

In CoqSQL, SQL methods are built on top *query plans*. A query plan represents a procedure to retrieve data from a database without causing mutation. The language construct is divided into five parts: expressions, projections, search terms, queries, and predicates. Figure 4-1 shows examples of relational algebra terms and query plans, with relation R having columns c_1, c_2, c_3 and relation S having columns c_4, c_5.

Relational Algebra Terms	**Query Plan**
$\sigma_{c_1=v}(R)$	QueryWhere (QueryTable R) (PredEq (ExprField c_1) (ExprValue v))
$\Pi_{c_1,c_5}(R \bowtie S)$	QueryProj (QueryJoin (QueryTable R) (QueryTable S)) (ProjCombine (ProjExpr (ExprField (CLeft c_1))) (ProjExpr (ExprField (CRight c_5))))
$\{c_2 + c_4 \| (c_1, c_2, c_3) \in R, (c_4, c_5) \in S\}$	QueryProj (QueryJoin (QueryTable R) (QueryTable S)) (ProjExpr (ExprBinary PLUS (ProjExpr (ExprField (CLeft c_2))) (ProjExpr (ExprField (CRight c_4)))))

Figure 4-1: Examples of translation from relational algebras to query plans.

Expressions An *expression* represents a scalar value under some context schema. An expression may be a projection of a field in the context schema, a concrete constant, or a result of applying one of the type universe's operators to sub-expressions.

```
Inductive Expr {TX : Types} : Schema → type → Type :=
| ExprField : ∀ {Γ : Tree type} {τ : type}, Choice Γ τ → Expr Γ τ
| ExprValue : ∀ {Γ : Schema} {τ : type}, ⟦τ⟧ → Expr Γ τ
| ExprConst : ∀ {Γ : Schema} {T : type}, constant T → Expr Γ T
| ExprUnary : ∀ {Γ : Schema} {S T : type}, unary S T → Expr Γ S → Expr Γ T
| ExprBinary : ∀ {Γ : Schema} {S T U : type},
               binary S T U → Expr Γ S → Expr Γ T → Expr Γ U
```

An expression `e : Expr Γ τ` is interpreted as a function from `e`'s context tuple (of type `⟨ Γ ⟩`) to a scalar value of type `⟦τ⟧`.

$$[\![\, \Gamma \vdash e : \tau \,]\!] : \langle\, \Gamma \,\rangle \to [\![\tau]\!]$$

```
Fixpoint denoteExpr {TX : Types} {Γ : Schema} {τ : type} (expr : Expr Γ τ)
  : ⟨ Γ ⟩ → ⟦τ⟧ :=
  match expr with
  | ExprField f ⇒ fun g ⇒ Choose f g
  | ExprValue v ⇒ fun g ⇒ v
  | ExprConst c ⇒ fun g ⇒ ⟦c⟧
  | ExprUnary f e ⇒ fun g ⇒ ⟦f⟧ (denoteExpr e g)
  | ExprBinary f e0 e1 ⇒ fun g ⇒ ⟦f⟧ (denoteExpr e0 g) (denoteExpr e1 g)
  end.

Instance DenotationExpr {TX : Types} {Γ : Schema} {τ : type}
  : Denotation (Expr Γ T) (⟨ Γ ⟩ → ⟦τ⟧) := {| denote := denoteExpr |}.
```

Projections A *projection* represents a conversion from one schema to another, similar to relational algebra's projection operation. The main difference is that CoqSQL's projection uses the schema's tree structure to specify the projection while relational algebra's projection uses attribute names.

```
Inductive Proj {TX : Types} : Schema → Schema → Type :=
| ProjCombine : ∀ {T U V : Schema}, Proj T U → Proj T V → Proj T (U ♦ V)
| ProjStar : ∀ {Γ : Schema}, Proj Γ Γ
| ProjLeft : ∀ {X Y} : Tree type, Proj (X ♦ Y) X
| ProjRight : ∀ {X Y} : Tree type, Proj (X ♦ Y) Y
| ProjExpr : ∀ {Γ : Schema} {T : type}, Expr Γ T → Proj Γ [T].
```

A projection `p : Proj Γ Γ'` is interpreted as a function from a tuple of type `⟨ Γ ⟩` to a tuple of type `⟨ Γ' ⟩`.

$$[\![\, p : \Gamma \Rightarrow \Gamma' \,]\!] : \langle\, \Gamma \,\rangle \to \langle\, \Gamma' \,\rangle$$

```
Fixpoint denoteProj {TX : Types} {Γ Γ' : Schema} (proj : Proj Γ Γ')
  : ⟨ Γ ⟩ → ⟨ Γ' ⟩ :=
  match proj with
  | ProjCombine p0 p1 ⇒ fun g ⇒ (denoteProj p0 g, denoteProj p1 g)
  | ProjStar ⇒ fun g ⇒ g
  | ProjExpr e ⇒ fun g ⇒ denoteExpr e g
  | ProjLeft ⇒ fst
  | ProjRight ⇒ snd
  end.

Instance DenotationProj {TX : Types} {Γ Γ' : Schema}
  : Denotation (Proj Γ Γ') (⟨ Γ ⟩ → ⟨ Γ' ⟩) :=
  {| denote := denoteProj |}.
```

Index Search Terms A *search term* encodes a list of expressions in a context schema to match a list of fields in a relation schema. There are two constructors.

- `SearchTermNil` is the base case where the list of fields is empty.
- `SearchTermCons` adds an expression and field pair on top of another index search term. Its matching condition is the conjunction of (1) the added expression-field comparison and (2) the matching result of the remaining fields.

```
Inductive SearchTerm {TX : Types} {σ : Schema} : Schema → list {Field σ} → Type :=
| SearchTermNil : ∀ Γ : Schema, SearchTerm Γ nil
| SearchTermCons : ∀ (Γ : Schema) (Fs : list (Field σ)) (F : Field σ),
                   Expr Γ F → SearchTerm Γ Fs → SearchTerm Γ (F :: Fs).
```

An index search term `m : SearchTerm (σ:=σ) Γ Fs` is interpreted as a function from `m`'s context tuple (of type `⟨ Γ ⟩`) to a Boolean predicate on the relation schema (of type `⟨ σ ⟩ → 𝔹`).

$$[\![\, \sigma, \Gamma, F_s \vdash m \,]\!] : \langle \, \Gamma \, \rangle \to \langle \, \sigma \, \rangle \to \mathbb{B}$$

```
Fixpoint denoteSearchTerm {TX : Types} {σ Γ : Schema} {Fs : list (Field σ)}
                          (st : SearchTerm Γ Fs)
  : ⟨ Γ ⟩ → ⟨ σ ⟩ → 𝔹 :=
  match st with
  | SearchTermNil ⇒ fun _ _ ⇒ true
  | SearchTermCons f e s ⇒
    fun g t ⇒ Choose f t =? denoteExpr e g && denoteSearchTerm s g t
  end%𝔹.

Instance DenotationSearchTerm {TX : Types} {σ Γ : Schema} {Fs : list (Field H)}
  : Denotation (SearchTerm Γ Fs) (⟨ Γ ⟩ → ⟨ σ ⟩ → 𝔹) :=
  {| denote := denoteSearchTerm |}.
```

Update Terms An *update term* represents a transformation of tuples within the same schema. Essentially, an update term encodes a list of fields and a list of expressions to replace those fields when transforming a tuple.

```
Inductive UpdateTerm {TX : Types} : Schema → Type :=
| UpdateTermNil : ∀ {σ}, UpdateTerm H
| UpdateTermCons : ∀ {σ} (F : Field σ), Expr σ F → UpdateTerm σ → UpdateTerm σ.
```

An update term `u : UpdateTerm σ` is interpreted as a function from a tuple of type `⟨ σ ⟩` to another tuple of the same type.

```
Fixpoint denoteUpdateTerm {TX : Types} {σ} (ut : UpdateTerm σ) : ⟨ σ ⟩ → ⟨ σ ⟩ :=
  match ut with
  | UpdateTermNil ⇒ fun t ⇒ t
  | UpdateTermCons f e u ⇒ fun t ⇒ Update f (denoteExpr e t) (denoteUpdateTerm u t)
  end.

Instance DenotationUpdateTerm {TX : Types} {σ : Schema}
  : Denotation (UpdateTerm σ) (⟨ σ → ⟨ σ ⟩) :=
```

```
{| denote := denoteUpdateTerm |}.
```

Queries and Predicate In CoqSQL, queries and predicates are defined as mutually inductive types. A *query* represents a data selection from a database. It is parameterized by two schemas: the context schema on which the query is performed, and the structure of the query's result. A *predicate* is parameterized by the schema on which the test is performed. In analogy to SQL, a predicate is what follows a WHERE clause.

```
Inductive Query {TX : Types} {DB : Database} : Schema → Schema → Type :=
| QueryTable : ∀ {Γ : Schema} {T : Table DB} (R := Pick T), Query Γ R
| QueryIndex : ∀ {Γ : Schema} {T : Table DB} (R := Pick T),
               ∀ idx : Index R, SearchTerm Γ (Pick idx) → Query Γ R
| QueryUnion : ∀ {Γ H : Schema}, Query Γ H → Query Γ H → Query Γ H
| QueryDistinct : ∀ {Γ H : Schema}, Query Γ H → Query Γ H
| QueryWhere : ∀ {Γ H : Schema}, Query Γ H → Pred (Γ ♦ H) → Query Γ H
| QueryJoin : ∀ {Γ X Y : Schema},
              Query Γ X → Query (Γ ♦ X) Y → Query Γ (X ♦ Y)
| QueryProj : ∀ {Γ X Y : Schema}, Query Γ X → Proj X Y → Query Γ Y
with Pred {X : Types} {DB : Database} : Schema → Type :=
| PredExists : ∀ {Γ σ : Schema}, Query Γ σ → Pred Γ
| PredEq : ∀ {Γ : Schema} {τ : type}, Expr Γ τ → Expr Γ τ → Pred Γ
| PredNeg : ∀ {Γ : Schema}, Pred Γ → Pred Γ
| PredAnd : ∀ {Γ : Schema}, Pred Γ → Pred Γ → Pred Γ
| PredOr : ∀ {Γ : Schema}, Pred Γ → Pred Γ → Pred Γ
| PredTrue : ∀ {Γ : Schema}, Pred Γ
| PredFalse : ∀ {Γ : Schema}, Pred Γ
| PredCast : ∀ {Γ Γ' : Schema}, Pred Γ → Proj Γ' Γ → Pred Γ'
```

A query `q : Query Γ σ` is interpreted as a function from `q`'s context tuple (of type $\langle\ \Gamma\ \rangle$) to a K-Relations representation of the result (of type $\langle\ \sigma\ \rangle \to \mathbb{N}$). A predicate `b : Pred Γ` is interpreted as a function from `b`'s context tuple to a Boolean.

$$[\![\ \Gamma \vdash q : \sigma\ ]\!] : \langle\ \Gamma\ \rangle \to \langle\ \sigma\ \rangle \to \mathbb{N}$$

$$[\![\ \Gamma \vdash b\ ]\!] : \langle\ \Gamma\ \rangle \to \mathbb{B}$$

```
Fixpoint denoteQuery {TX : Types} {DB : Database} (state : SQLState DB)
          {Γ σ : Schema} (query : Query (DB:=DB) Γ σ) : 〈 Γ 〉 → 〈 σ 〉 → ℕ :=
  match query with
  | QueryTable tb ⇒ fun g t ⇒ state tb t
  | QueryIndex tb idx st ⇒ fun g t ⇒ ‖denoteSearchTerm st g t‖ × state tb t
  | QueryUnion s0 s1 ⇒ fun g t ⇒ denoteQuery state s0 g t + denoteQuery state s1 g t
  | QueryDistinct s ⇒ fun g t ⇒ ‖0 <? denoteQuery state s g t‖
  | QueryWhere s p ⇒ fun g t ⇒ ‖denotePred state p (g, t)‖ × denoteQuery state s g t
  | QueryJoin s0 s1 ⇒
      fun g t ⇒ denoteQuery state s0 g (fst t) ×
                denoteQuery state s1 (g, fst t) (snd t)
  | QueryProj s p ⇒
      fun g t ⇒ fold (fun t' acc ⇒
                        acc + ‖t =? denoteProj p t'‖ × (denoteQuery state s g t')) 0
  end
with denotePred {TX : Types} {DB : Database} (state : SQLState DB)
                {Γ : Schema} (pred : Pred (DB:=DB) Γ) : 〈 Γ 〉 → 𝔹 :=
  match pred with
  | PredExists s ⇒
      fun g ⇒ fold (fun t acc ⇒ acc || (0 <? denoteQuery state s g t)) false
  | PredEq e0 e1 ⇒ fun g ⇒ denoteExpr e0 g =? denoteExpr e1 g
  | PredNeg p ⇒ fun g ⇒ negb (denotePred state p g)
  | PredAnd p0 p1 ⇒ fun g ⇒ denotePred state p0 g && denotePred state p1 g
  | PredOr p0 p1 ⇒ fun g ⇒ denotePred state p0 g || denotePred state p1 g
  | PredTrue ⇒ fun _ ⇒ true
  | PredFalse ⇒ fun _ ⇒ false
  | PredCast pred proj ⇒ fun g ⇒ denotePred state pred (denoteProj proj g)
  end%𝔹.
```

4.4 SQL Methods

An *SQL method* corresponds to an SQL statement in conventional SQL. CoqSQL models SQL methods on top of query plans. The following is the inductive definition of SQL methods, consisting of select, insert, delete, and update clauses.

```
Inductive SQLMethod (TX : Types) (DB : Database) : Type → Type :=
| SQLSelect : ∀ {σ : Schema}, Query [unit_t] σ → SQLMethod DB (list 〈 σ 〉)
| SQLInsert : ∀ (T : Table DB) (R := Pick T), 〈 R 〉 → SQLMethod DB unit
| SQLDelete : ∀ (T : Table DB) (R := Pick T), Pred R → SQLMethod DB unit
| SQLUpdate : ∀ (T : Table DB) (R := Pick T),
                UpdateTerm R → Pred R → SQLMethod DB unit.
```

An SQL method acts as a label in the transition system. We define an evaluation

step relation for our SQL specification that shows possible transitions, given an SQL method, from an SQL state to another state with a return result.

```
Inductive SQLEval {TX : Types} (DB : Database)
  : ∀ {Ret}, SQLMethod DB Ret → SQLState DB → Ret → SQLState DB → Prop :=
| SQLSelectEval :
    ∀ H st (query : Query [unit_t] H) ret,
      (∀ t, count_occ eq_dec ret t = denoteQuery st query tt_t t)
      → SQLEval DB (SQLSelect DB query) st ret st
| SQLInsertEval :
    ∀ tb tp st,
      SQLEval DB (SQLInsert DB tb tp) st tt
        (overrideDep eq_dec st tb (fun t ⇒ ‖tp =? t‖ + st tb t))
| SQLDeleteEval :
    ∀ tb pred st,
      SQLEval DB (SQLDelete DB tb pred) st tt
        (overrideDep eq_dec st tb (fun t ⇒ ‖negb (denotePred st pred t)‖ × st tb t))
| SQLUpdateEval :
    ∀ tb ut pred st,
      SQLEval DB (SQLUpdate DB tb ut pred) st tt
        (overrideDep eq_dec st tb
          (fun t ⇒ fold (fun t' acc ⇒ acc + ‖ if denotePred st pred t'
                                                then t =? denoteUpdateTerm ut t'
                                                else t =? t' ‖ × st tb t') 0)).
```

4.5 SQL Specification Objects

At this point, an SQL specification object for a database can be created using `SQLState` as the state and `SQLEval` as the transition relation. This object represents the correct behavior that SQL implementations must follow.

```
Definition SQLObject {TX : Types} (DB : Database) : Object (SQLMethod DB) :=
  {|
    State := SQLState DB;
    EvalInstruction := @SQLEval TX DB;
  |}.
```

To achieve a transactional SQL semantics, a programmer first declares a database structure and a program for high-level methods on top of the SQL methods from that database.

```
Example ExampleDatabase : Database. □
Example ExampleMethod : Type → Type. □
Example ExampleProgram
  : ∀ Ret, ExampleMethod Ret → Program (SQLMethod database) Ret. □
```

After that, an abortable implementation and its initial state for the methods must be created. Any strategy to build the implementation is acceptable. In Chapter 6, we will present a library that automatically creates an implementation for users.

```
Example ExampleImplementation : Implementation (Abortable ExampleMethod). □
Example ExampleInitialState : ExampleImplementation.(implBase).(State). □
```

This implementation can be extracted to an executable program via the Transactions framework's extraction mechanism. To ensure that the implementation is correct, the developer must prove that the thread behavior of the implementation can be simulated by the transactional semantics of the SQL specification object's serial implementation. This proof is automatically done if the implementation is synthesized via CoqSQL's library.

```
Example ExampleImplementationCorrect
  : Simulates
      (ThreadsBehavior ExampleImplementation)
      (superposition
        (ThreadsBehavior
           (serialImplementation
              (abortableObject
                 (methodObject
                    (programObject (SQLObject ExampleDatabase)) ExampleProgram)))))
      (pair nil ExampleInitialState)
      (eq (pair nil (SQLStateInit ExampleDatabase))). ■
```

Chapter 5

Implementations

In addition to the complete specifications, CoqSQL comes with naive implementations of SQL databases. This chapter discusses the strategy that CoqSQL uses to systematically synthesize the SQL database implementation of any database schema with the proof of correctness.

5.1 SQL Table Specification Objects

To modularize the synthesis process, we introduce *SQL table methods* and *SQL table specification objects*. They represent low-level database operations and their semantics. These operations are significantly simpler than ordinary SQL statements. For instance, `SQLTableDelete` directly takes a function from a tuple to a Boolean as the predicate. `SQLTableSelectIndex` takes a search term (of type `SearchTerm Γ (Pick idx)`) and a context tuple (of type `⟨ Γ ⟩`) that contains free variables in the search term. CoqSQL first synthesizes the implementation of the low-level SQL table methods. The synthesis results are later used to build the implementation of the higher-level SQL methods.

```
Inductive SQLTableMethod {TX : Types} (DB : Database) : Type → Type :=
| SQLTableSelectTable :
    ∀ (T : Table DB) (R := Pick T), SQLTableMethod DB (list ⟨ R ⟩)
```

```
| SQLTableSelectIndex :
    ∀ {Γ : Schema} (T : Table DB) (R := Pick T) (idx : Index R),
      ⟨ Γ ⟩ → SearchTerm Γ (Pick idx) → SQLTableMethod DB (list ⟨ R ⟩)
| SQLTableInsert :
    ∀ (T : Table DB) (R := Pick T), ⟨ R ⟩ → SQLTableMethod DB unit
| SQLTableDelete :
    ∀ (T : Table DB) (R := Pick T), (⟨ R ⟩ → 𝔹) → SQLTableMethod DB unit
| SQLTableUpdate :
    ∀ (T : Table DB) (R := Pick T), (⟨ R ⟩ → ⟨ R ⟩) → SQLTableMethod DB unit.

Definition SQLTableObject {TX : Types} (DB : Database) : Object (SQLTableMethod DB) :=
  {|
    State := SQLState DB;
    EvalInstruction := @SQLTableMethodEval TX DB
  |}.
```

5.2 SQL Table Implementations

Recall that the SQL database structure is a tree of relations. We break the process of building the implementation of SQL table methods into two cases: the base case where the database is a singleton tree, and the inductive case where the database is a tree consisting of two sub-databases.

5.2.1 Single Table Implementation

We use a list of tuples as the concrete representation of a singleton database. The Transactions framework's map structure (from unit to a list of tuples) is used as the base structure. Note that this is a degenerate use of maps as registers.

```
Module Unit.
  Definition t := unit.
  Definition eq_dec : ∀ p q : unit, {p = q} + {p ≠ q}. □
End Unit.
Module UnitPMap := PMapFun Unit.
```

Using a list as the internal representation, implementing a program for SQL table methods is straightforward. Relevant details are given below. At the end, we prove that the implementation object simulates the SQL table specification object.

```
Section SQLTableList.
  Context {TX : Types}
          (R : Relation).

  Definition ListPStruct := UnitPMap.pStruct.

  Definition ListImplProg {Ret} (method : SQLTableMethod [R] Ret)
    : Program (Map.MapMethod unit (list ⟨ R ⟩)) Ret :=
    match method with
    | SQLTableSelectTable _ _ _ ⇒ tps ← Map.get tt;
                                   Return (list_or_nil tps)
    | SQLTableInsert _ _ t ⇒ tps ← Map.get tt;
                             _ ← Map.put tt (t :: (list_or_nil tps));
                             Return tt
    (* More cases *) ...
    end.

  Definition ListImplObject : Object (SQLTableMethod [R]). □
  Definition ListImplInit : ListImplObject.(State). □
  Theorem ListImpl_ok :
    Simulates ListImplObject (SQLTableObject [R]) ListImplInit (SQLStateInit [R]). ■
End SQLTableList.
```

5.2.2 Table Implementation Composer

For the inductive case, we build a more-complicated database implementation using the Transactions framework's pair combinator on top of SQL table methods of the sub-databases. Logically, the implementation of the composed database simply dispatches the method calls to the proper lower-level interfaces. The only nontrivial part is to convince Coq's type checker that the types of method calls' arguments match appropriately with the types of lower-level interfaces.

```
Section SQLTableProduct.
  Context {TX : Types}.
          (DL DR : Database).

  Definition ProdImplProg {Ret} (method : SQLTableMethod (DL ♦ DR) Ret)
    : Program (PairMethod (SQLTableMethod DL) (SQLTableMethod DR)) Ret.
  Proof.
    match method with
    | SQLTableSelectTable _ tb ⇒
      match tb in Path schm
```

```
          return (match schm return Path schm → _ with
                  | [ _ ] ⇒ fun _ ⇒ IDProp
                  | u ⧫ v ⇒ fun tb' ⇒ Program (_ u v) (list ⟨ Pick tb' ⟩)
                  end tb) with
    | PHere ⇒ idProp
    | PLeft tb' ⇒ ret ← inl (SQLTableSelectTable _ tb');
                  Return ret
    | PRight tb' ⇒ ret ← inr (SQLTableSelectTable _ tb');
                   Return ret
    end
  (* More cases *) ...
  end.

  Definition ProdImplObject : Object (SQLTableMethod (DL ⧫ DR)). □
  Definition ProdImplInit : ProdImplObject.(State). □
  Lemma ProdImpl_ok :
    Simulates
      ProdImplObject (SQLTableObject (DL ⧫ DR))
      ProdImplInit (SQLStateInit (DL ⧫ DR)). ■
End SQLTableProduct.
```

5.3 Naive Database Engine

Next, we create an implementation object for SQL methods on top of SQL table methods. We call such implementation a *database engine*. CoqSQL includes a naive database engine that performs basic list operations corresponding to SQL query plans and properly calls SQL table methods for low-level operations.

```
Section NaiveEngine.
  Context {TX : Types}.
          (DB : Database).

  Fixpoint EngineImplProgQuery {Γ σ} (ctx : ⟨Γ⟩) (query : Query Γ σ)
    : Program (SQLTableMethod DB) (list ⟨σ⟩) :=
    match query in Query Γ σ return ⟨Γ⟩ → Program (SQLTableMethod DB) (list ⟨σ⟩) with
    | QueryTable tb ⇒ fun c ⇒ r ← SQLTableSelectTable _ tb;
                              Return r
    | QueryIndex tb idx st ⇒ fun c ⇒ r ← SQLTableSelectIndex _ tb idx c st;
                                     Return r
    | QueryUnion s0 s1 ⇒ fun c ⇒ r0 ⟵ EngineImplProgQuery c s0;
                                 r1 ⟵ EngineImplProgQuery c s1;
                                 Return (r0 ++ r1)
    (* More cases *) ...
```

```
    end ctx
  with EngineImplProgPred {Γ} (ctx : ⟨Γ⟩) (pred : Pred Γ)
    : Program (SQLTableMethod DB) 𝔹 :=
    match pred with
    | PredExists s ⇒ fun c ⇒ r ⟵ EngineImplProgQuery c s;
                             Return (negb (r =? nil))
    | PredEq e0 e1 ⇒ fun c ⇒ Return (denoteExpr e0 c =? denoteExpr e1 c)
    (* More cases *) ...
    end ctx.

  Definition NaiveEngineImplProg {Ret} (method : SQLMethod DB Ret)
    : Program (SQLTableMethod DB) Ret :=
    match method with
    | SQLSelect _ sel ⇒ EngineImplProgQuery (Γ := [unit_t]) tt_t sel
    | SQLDelete _ tb pred ⇒
      tps ⟵ EngineImplProgQuery (Γ := [unit_t]) tt_t
                (QueryWhere (QueryTable tb) (PredCast pred ProjRight));
      _ ← SQLTableDelete _ tb (fun t ⇒ if in_dec eq_dec t tps then true else false);
      Return tt
    (* More cases *) ...
    end.

  Definition NaiveEngineImplObject : Object (SQLMethod DB).
  Definition NaiveEngineImplInit : NaiveEngineImplObject.(State).
  Theorem NaiveEngineImpl_ok :
    Simulates
      NaiveEngineImplObject (SQLObject DB) NaiveEngineImplInit (SQLStateInit DB). ■
End NaiveEngine.
```

5.4 Synthesis Module

The last step is to glue together the whole pipeline to create an SQL implementation of the base machine interface. We use `flattenPStruct` and `mapPStruct` combinators to build the predicated structure for any database schema. It follows that the predicated structure's behavior simulates the SQL specification object.

```
Fixpoint SQLVal {TX : Types} (db : Database) : Type :=
  match db with
  | [ r ] ⇒ list ⟨ r.(heading) ⟩
  | tl ⧫ tr ⇒ sum (SQLVal tl) (SQLVal tr)
  end.

Fixpoint SQLTablePStruct {TX : Types} (db : Database)
```

```
  : PStruct (SQLTableMethod db) (SQLVal db) :=
  match db with
  | [ r ] ⇒ flattenPStruct (@ListImplProg _ r) (ListPStruct (SQLVal [r]))
  | tl ⧫ tr ⇒ flattenPStruct (@ProdImplProg _ tl tr)
                  (pairPStruct (mapPStruct inlInjection (SQLTablePStruct tl))
                               (mapPStruct inrInjection (SQLTablePStruct tr)))
  end.

Definition SQLPStruct {TX : Types} (db : Database)
  : PStruct (SQLMethod db) (SQLVal db) :=
  flattenPStruct (@BasicEngineImplProg _ _) (SQLTablePStruct db).

Lemma SQLPStructSim {TX : Types} (db : Database)
  : Simulates (SQLPStruct db).(pStructSpec) (SQLObject db)
              (SQLPStruct db).(pStructSpecInit) (SQLStateInit db). ■
```

The `EngineArgs` module type encapsulates the necessary information for a database including the schema, the methods, and the abstract implementation. The `EngineDatabase` module, given an instance of `EngineArgs`, creates an extractable SQL implementation with its linearizabilty proof. At this point, users can use this module to synthesize a naive database implementation only by giving a database structure and a program implementation on top of the SQL abstract methods!

```
Module Type EngineArgs.
  Parameter TX : Types.
  Parameter database : Database.
  Parameter method : Type → Type.
  Parameter implProg : ∀ Ret, method Ret → Program (SQLMethod database) Ret.
End EngineArgs.

Module EngineDatabase (E : EngineArgs).
  ...
  ...
  Definition pStruct := SQLPStruct database.
  ...
  Definition Impl : Implementation (Abortable method) :=
    mapImplementation (mapAbortable implProg) (tStructImpl pStruct tm).
  ...
  Lemma Impl_ok :
    Simulates
      (ThreadsBehavior Impl)
      (superposition
        (ThreadsBehavior
```

```
          (serialImplementation
            (abortableObject (methodObject (programObject (SQLObject _)) implProg)))))
      (nil, tStructInitState pStruct tm tmInit)
      (eq (nil, SQLStateInit _)). ■
End EngineDatabase.
```

It is important to note that the index structures are not used in our naive database engine. It is future work to create a more optimized verified engine that takes advantages of indexes (see Section 7.2).

Chapter 6

Improving CoqSQL's Usability

This chapter discusses an attempt to improve CoqSQL's usability for ordinary users. At the end, we evaluate CoqSQL by showing a number of complicated use cases that the framework is able to handle.

6.1 Named-to-Unnamed Translation

CoqSQL uses the unnamed approach to represent SQL schemas to avoid naming collisions. Although such a representation simplifies SQL formalizations, it makes writing SQL statements harder for human programmers. The section presents a more SQL-like language that can be translated to the unnamed SQL presented in the previous chapters. The translation process is completely unverified, and its sole purpose is to improve CoqSQL's usability.

6.1.1 Bounded Index Structure

Inspired by Fiat [4], CoqSQL uses the bounded index structure to encapsulate a position in a vector. The idea is that the value at the position is encoded in the bounded index structure. With some custom Ltac, we can make Coq's typeclass resolution automatically construct an `IndexBound` property of a `BoundedIndex` by searching through the vector to find the position at which the value matches `bindex`.

```
Class IndexBound {A : Type} (n : ℕ) (a : A) (Bound : Vector.t A n) :=
  {
    ibound : Fin.t n;
    boundi : Bound[@ibound] = a;
  }.

Class BoundedIndex {A : Type} (n : ℕ) (Bound : Vector.t A n) :=
  {
    bindex : A;
    indexb : IndexBound bindex Bound
  }.

Notation "`` idx" := ({| bindex := idx |}) (at level 0).
```

6.1.2 Indexed Vector

Next, we define the indexed vector structure. An `iVector` encapsulates a vector of values (`vecVals`) and a vector of their names (`vecNames`). This abstraction allows any position in the vector to be referred to the string value in `vecNames` . We define `iVectorIndex` as a datatype for storing some position in an indexed vector. `iVectorAt` projects the value at the corresponding position in `vecVals` .

```
Record iVector (A : Type) :=
  {
    vecSize : ℕ;
    vecNames : Vector.t string vecSize;
    vecVals : Vector.t A vecSize;
  }.

Definition iVectorIndex {A} (v : iVector A) :=
  BoundedIndex v.(vecNames).

Definition iVectorAt {A} {v : iVector A} (index : iVectorIndex v) :=
  v.(vecVals)[@index.(indexb).(ibound)].
```

We can "name" every element in a Coq vector `v : Vector.t A` by building an indexed vector that has `v` as its `vecVals` . Thanks to Coq's typeclass resolution, a position in the indexed vector can be constructed from a string that matches some member of `vecNames` . After that, the element in `vecVals` with the exact same position can be projected using `iVectorAt` . Following is an example usage of indexed vectors.

```
Definition SampleVecVals : Vector ℕ := [1; 2; 3; 4].
Definition SampleVecNames : Vector string := ["A"; "B"; "C"; "D"].

Definition SampleIVector : iVector ℕ :=
  {|
    vecSize := 4;
    vecNames := SampleVecNames;
    vecVals := SampleVecNames;
  |}.

Definition SampleIndex : iVectorIndex SampleIVector := ``"B".
Eval simpl in (iVectorAt SampleIVector SampleIndex)
(*  = 2
    : ℕ
*)
```

6.1.3 Vector-to-Tree Conversion

An indexed vector can be easily translated to a right-heavy tree as shown below. Note that `iVectorTree` takes a default value `e` in case the input vector is empty.

```
Fixpoint iVectorTree' {A n} (e : A) (v : Vector.t A n) : Tree A :=
  match v with
  | [] ⇒ [ e ]
  | Vector.cons _ h _ [] ⇒ [ h ]
  | Vector.cons _ h _ v' ⇒ [ h ] ⧫ iVectorTree' e v'
  end.

Definition iVectorTree {A} (e : A) (v : iVector A) : Tree A :=
  iVectorTree' e v.(vecVals).
```

6.1.4 Named Data Representations

We finish this section by showing the named SQL data representations that CoqSQL uses. The idea is to combine the original tree structure with indexed vectors to build richer structures that allow uses to specify everything by names.

Schemas A schema for the named representation is a record consisting of the original schema structure and an indexed vector of the schema's fields.

```
Record NSchema {TX : Types} :=
  {
    rTree : Schema;
    rFields : iVector (Field rTree);
  }.
```

Relations A relation consists of a named schema and a tree of indexes. Note that we do have names for indexes because conventional SQL statements do not include indexes.

```
Record NRelation {TX : Types} :=
  {
    rState : NSchema;
    rIndexes : Tree (list (Field rState.(rTree)));
  }.
```

Databases A database consists of a tree structure of relations and an indexed vector to use for mapping a table name to the path in the tree structure.

```
Record NDatabase {TX : Types} :=
  {
    rRelTree : Tree NRelation;
    rPaths : iVector (Path rRelTree);
  }.
```

SQL statements for the named representations are roughly similar to what was presented in Chapter 4. We skipped the discussion to avoid redundancy in this thesis. We also define a number of Coq notations to make writing SQL statements intuitive. Their usages are shown in Section 6.3.

6.2 Simple SQL Type Universe

CoqSQL comes with a type universe instance that encapsulates a subset of SQL types. It consists of four standard types, including `UNIT`, `BOOL`, `UINT`, and `VARCHAR n`, and the proof that all the types are finite and decidable.

```
Inductive SQLType := UNIT | BOOL | UINT | VARCHAR (n : ℕ).

Instance denotationSQLType : Denotation SQLType Type :=
  {
    denote := fun t ⇒ match t with
                      | UNIT ⇒ unit
                      | BOOL ⇒ 𝔹
                      | UINT ⇒ { n : N | n <= 2 ^ 32 - 1 }
                      | VARCHAR n ⇒ { s : string | length s <= n. }
                      end
  }.

Instance finiteType : ∀ τ : SQLType, Finite ⟦τ⟧. □
Instance decidableType : ∀ τ : SQLType, Decidable ⟦τ⟧. □
```

In addition, the type universe contains a number of SQL binary operations, including `PLUS`, `MULT`, `MINUS`, and `ISLE`. For arithmetic operations, the result value wraps around upon reaching 2^{32}. The preloaded SQL type universe neither includes constant nor unary operations, but extending it to support more operations is straightforward for users.

```
Instance denotationBinary S T U
  : Denotation (SQLBinary S T U) (⟦S⟧ → ⟦T⟧ → ⟦U⟧) :=
  {
    denote := fun c ⇒ match c with
                      | PLUS ⇒ fun e1 e2 ⇒ n_to_UINT (proj1_sig e1 + proj1_sig e2)
                      | MULT ⇒ fun e1 e2 ⇒ n_to_UINT (proj1_sig e1 × proj1_sig e2)
                      | SUB ⇒ fun e1 e2 ⇒ n_to_UINT (proj1_sig e1 - proj1_sig e2)
                      | ISLE ⇒ fun e1 e2 ⇒ proj1_sig e1 ≤? proj1_sig e2
                      end
  }.

Instance SQLTypes : Types :=
  {
    type := SQLType;
    denotationType := denotationSQLType;
    ...
    ...
  }.
```

6.3 Evaluation

We evaluate CoqSQL through a few interesting use cases to show that the framework is capable of handling complicated SQL queries.

6.3.1 Bookstore Database

We first revisit the example presented in the introduction. The database consists of three tables (`"ACCOUNT"`, `"BOOK"`, and `"ORDER"`) and five methods. We discuss the interesting features that the implementation uses in detail.

```
Module BookStoreDatabaseArgs.
  ...
  ...

  Definition implProg Ret (m : method Ret) : Program (SQLMethod database) Ret :=
    match m with
    ...
    ...
    | placeOrder id isbn ⇒
      bal ← QUERY SELECT VAR("a.Balance") AS "bal"
                  FROM "ACCOUNT" AS "a" WHERE VAR("a.Id") == VAL(id);
      price ← QUERY SELECT VAR("b.Price") AS "price"
                    FROM "BOOK" AS "b" WHERE VAR("b.Isbn") == VAL(isbn);
      match bal, price with
      | [ bal' ], [ price' ] ⇒
        if price' ≤? bal' : 𝔹
        then _ ← INSERT INTO "ORDER" VALUES (id ,, isbn);
             _ ← UPDATE "ACCOUNT" SET "Balance" = VAR("Balance") - VAL(price')
                 WHERE VAR("Id") == VAL(id);
             Return true
        else Return false
      | _, _ ⇒ Return false
      end
    | removeAccount id ⇒
      _ ← DELETE FROM "ACCOUNT" WHERE VAR("Id") == VAL(id);
      _ ← DELETE FROM "ORDER" WHERE VAR("AccountId") == VAL(id);
      Return tt
    end.
End BookStoreDatabaseArgs.
```

placeOrder involves a consistency check between values from two separate queries before calling two atomic SQL operations (one insert clause and one update clause). The transaction semantics guarantees that either (1) both an order is placed and the balance is deducted or (2) none of them is completed.

removeAccount shows that the framework is able to handle atomic deletion. In other words, other SQL executions cannot interleave between the two delete clauses. Thus, it is guaranteed that if an account is removed, other SQL queries must not see any order associated with that account.

6.3.2 School Database

Another example is a database for a school's office of the registrar. Users use the database to store information about students, faculty, and courses. It involves a numbers of nontrivial SQL queries. We use this example to show that CoqSQL is able to handle complicated SQL queries.

```
Module SchoolDatabaseArgs.
  ...
  ...

  Definition schema :=
    DATABASE ⟨
      TABLE "STUDENT" ⟨ "Id" ::: UINT; "Name" ::: VARCHAR 20 ⟩;
      TABLE "FACULTY" ⟨ "Id" ::: UINT; "Name" ::: VARCHAR 20 ⟩;
      TABLE "COURSE" ⟨ "Id" ::: UINT; "FacultyId" ::: UINT ⟩;
      TABLE "REGISTRATION" ⟨ "StudentId" ::: UINT; "CourseId" ::: UINT ⟩
    ⟩.
  ...
  ...

  Definition implProg Ret (m : method Ret) : Program (SQLMethod database) Ret :=
    match m with
    ...
    ...
    | allNames ⇒
      r ← QUERY
            SELECT DISTINCT
             (SELECT VAR("s.Name") AS "name" FROM "STUDENT" AS "s")
            UNION
```

```
            (SELECT VAR("f.Name") AS "name" FROM "FACULTY" AS "f");
      Return r
    | studentsOfFaculties facultyName ⇒
      r ← QUERY
            SELECT DISTINCT
              SELECT VAR("r.StudentId") AS "id"
              FROM "REGISTRATION" AS "r"
              JOIN "COURSE" AS "c" JOIN "FACULTY" AS "f"
              WHERE VAR("r.CourseId") == VAR("c.Id")
              AND VAR("c.FacultyId") == VAR("f.Id")
              AND VAR("f.Name") == VAL(facultyName);
      Return r
    | teachingFaculties ⇒
      r ← QUERY
            SELECT VAR("f.Name") AS "name"
            FROM "FACULTY" AS "f"
            WHERE (EXISTS SELECT * FROM "COURSE" AS "c"
                          WHERE VAR("c.FacultyId") == VAR("f.Id"));
      Return r
    end.
End SchoolDatabaseArgs.
```

allNames shows the framework's ability to synthesize queries involving DISTINCT and UNION clauses. Note that it successfully merges the query results from two different relations.

studentsOfFaculties involves joining *three* relations to find relevant information. The SQL query involves two comparisons of tuples from different relations and one comparison with an outside parameter.

teachingFaculties uses an SQL EXISTS predicate where the condition inside the predicate involves the outer query result.

Chapter 7

Related Work and Future Research

7.1 Related Work

Verified Relational Database Systems Malecha et al. [7] introduced a fully verified relational database implementation in Coq with support for efficient data models, including B+Trees, and query optimization via runtime cost estimation. Although the system supports most SQL features, it lacks the "ACID" guarantee, which is essential in any concurrent system. Additionally, the system is not primarily designed for extensibility and ease of usage. CoqSQL aims to solve both of the problems (minus the "D" durability part of "ACID") while still providing a similar formal correctness guarantee.

SQL Formal Semantics There exist a number of studies in SQL formal semantics, both from the formal-methods community and the database-theory community. One approach is to model an SQL relation as a list of tuples [7, 10]. An SQL query becomes a function that returns a list, and two queries are equivalent if one's result is a permutation of the other's. Although this approach is intuitive, proving correctness under this semantics is often lengthy and difficult. Another approach, K-Relations [5], represents an SQL relation as a function mapping from a tuple to the number of its occurrences. However, in order to have K-Relations semantics for projections, the set

of all possible tuples must be finite. A workaround has been introduced in HottSQL [2], which uses univalent types as the values of the function instead of natural numbers. While this allows the semantics to support projections without a finiteness requirement via Sigma types, it is not directly computable, and is useful only for proving theorems. CoqSQL chooses to adopt K-relations as well, but explicitly requires the proof of finiteness for all SQL types from users.

Proof-Guided Deductive Synthesis Deductive synthesis [8] is a technique used for synthesizing a computer program from a given specification. Starting from a declarative goal, the system converts the goal into a theorem-proving task. The task can be solved using a set of deductive rules. The synthesis process finishes once the theorem is successfully proved. CoqSQL's synthesis work is mainly inspired by Fiat [4], a deductive system that allows users to synthesize programs from declarative specifications with a high level of automation. Users start with an expressive specification of a program and repeatedly refine the specification until they reach a concrete implementation. Each refinement step leaves a proof trace showing its correctness. In CoqSQL, the framework starts from a high-level specification of the implementation via the Transactions Framework's abstract object. The framework then stepwisely synthesizes the implementation as it progresses through proving SQL correctness theorems.

7.2 Future Research

Overall this research has achieved the goal set at the beginning of the project. However, there is some room for improvement that we noticed while developing CoqSQL.

More SQL Features CoqSQL supports most SQL features, but not all. In particular, it would be useful to make CoqSQL support SQL aggregation. One possible idea is to include aggregator operations into the type universe typeclass.

```
Class Types :=
  {
    ...
    ...
    aggregator : type → type → Type;
    denotationAggregator : ∀ S T, Denotation (aggregator S T) ((⟦S⟧ → ℕ) → ⟦T⟧);
  }.
```

This change makes adding aggregation to our SQL specification trivial. However, unlike other operations (constant, unary, and binary), the implementation of aggregation is not as simple because it involves iterating through a concrete SQL representation (which is not a function from a tuple to a natural number). Furthermore, CoqSQL would be even more useful if it supported the GROUP BY operation. HottSQL's specification desugars a GROUP BY clause into a complex nested SELECT clause. Unfortunately, we have not spent much time investigating it.

More Efficient Implementations The library included in CoqSQL only provides users with naive database implementations. While the provided list implementations are correct, they are arguably inefficient. Modern database systems often use complicated structures such as B+Trees or hash tables to manage data. Therefore, one important improvement for CoqSQL is to include those structures in the library. It would be an interesting research problem to make the framework able to synthesize and choose the most efficient data structure for a database's representation given that the information about the SQL statements that will be executed and the set of indexes that will be used are known beforehand.

SQL Optimization Flow As of now, users are able to show that two SQL statements are equivalent by proving that their denotations are equivalent. It would be useful if CoqSQL automatically rewrote SQL queries for users to increase their efficiency. One major improvement is to make the framework pick the correct index for an SQL query that involves selecting every row in a table and filtering the result with some predicate.

Bibliography

[1] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. "Transactional predication: high-performance concurrent sets and maps for STM". In: *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM. 2010, pp. 6–15.

[2] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. "HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics". In: *Proc. PLDI*. 2017.

[3] Oracle Corporation. *MySQL Bugs: Statistics*. URL: `https://bugs.mysql.com/tide.php` (visited on 04/12/2017).

[4] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. "Fiat: Deductive synthesis of abstract data types in a proof assistant". In: *ACM SIGPLAN Notices*. Vol. 50. 1. ACM. 2015, pp. 689–700.

[5] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. "Provenance Semirings". In: *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '07. Beijing, China: ACM, 2007, pp. 31–40. ISBN: 978-1-59593-685-1. DOI: `10.1145/1265530.1265535`.

[6] PostgreSQL Global Development Group. *PostgreSQL Mailing Lists: psql-bugs*. URL: `https://www.postgresql.org/list/pgsql-bugs/` (visited on 04/12/2017).

[7] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. "Toward a verified relational database management system". In: *ACM Sigplan Notices*. Vol. 45. 1. ACM. 2010, pp. 237–248.

[8] Zohar Manna and Richard Waldinger. "A Deductive Approach to Program Synthesis". In: *ACM Trans. Program. Lang. Syst.* 2.1 (Jan. 1980), pp. 90–121. ISSN: 0164-0925. DOI: `10.1145/357084.357090`.

[9] The Coq development team. *The Coq Proof Assisant*. URL: https://coq.inria.fr/distrib/current/refman/ (visited on 04/12/2017).

[10] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. "Qex: Symbolic SQL Query Explorer". In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR'10. Dakar, Senegal: Springer-Verlag, 2010, pp. 425–446. ISBN: 3-642-17510-4, 978-3-642-17510-7. URL: http://dl.acm.org/citation.cfm?id=1939141.1939165.