Formal Verification of Relational Algebra Transformations in Fiat2 Using Coq

by

Christian Teshome

B.S Electrical Engineering and Computer Science, MIT, 2024

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

May 2025

Authored by:	Christian Teshome Department of Electrical Engineering and Computer Science May 16, 2025
Certified by:	Adam Chlipala Professor of Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Formal Verification of Relational Algebra Transformations in Fiat2 Using Coq

by

Christian Teshome

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Data-intensive applications often involve operations over structured datasets, such as filtering, joining, and projecting records. Relational database systems generally use query planners to optimize high-level SQL queries into efficient execution plans. While these systems apply well-established query transformations, they typically assume the correctness of these transformations rather than formally proving them. The absence of formal guarantees can be a significant limitation for systems with strict correctness requirements.

This thesis contributes to Fiat2, a Python-like high-level programming language for data-intensive workloads that integrates formal verification via the Coq proof assistant. We focus on proving the correctness of several rewrite-based query optimizations commonly used in database engines. Specifically, we formalize and prove the correctness of algebraic rewrites involving combinations of filters, joins, and projections, as well as join-reordering rewrites.

All rewrites are proven in Coq to preserve the semantics of the original program under list semantics, meaning that the output lists are fully equivalent (or permutations, in the case of join reordering). These verified rewrites serve as a foundation for future optimization in Fiat2, enabling significant optimizations while preserving the semantics of the original queries with correctness guarantees. The results demonstrate the feasibility of integrating formally verified query optimizations into a practical high-level programming language.

Thesis supervisor: Adam Chlipala
Title: Professor of Computer Science

Acknowledgments

I would like to begin by thanking my advisor, Adam Chlipala, for his support and guidance throughout the project, from introducing me to Fiat2 to providing detailed feedback and examples at every stage. His advice was instrumental in shaping the direction of my work.

I would also like to thank Dustin Jamner and Xin Zhang for all their help and mentorship throughout the project. Before working on this project, I had very little experience with theorem proving and proof writing in general. Both Dustin and Xin have been meeting regularly with me and patiently helping me figure out the intricacies of Coq, how to tackle difficult proofs, how to formulate theorem statements correctly, etc. They have always provided good advice and helped me slowly build up intuition for theorem proving. I would also like to thank Jack Feser for joining our weekly Fiat2 meetings remotely every week and providing his unique perspective on database optimizations and helpful suggestions regarding the project.

Also, I was funded via TAship for 6.101 / 6.009 Fundamentals of Programming this spring, so I would also like to thank the 6.101 course staff for giving me the opportunity to work with them and have an impact on the course this semester. For every semester I have been part of 6.101 staff, it has always been a great experience. Finally, I would like to thank the MIT Lion Dance Team for being a great and fun community to be a part of this year.

Contents

List of Figures

Chapter 1

Introduction

In recent years, formal verification has made significant progress towards achieving fully verified computing systems, where all aspects of a computing stack are formally proven correct, including the processors, compilers, and high-level applications. These verified stacks are invaluable for applications where having 100% correctness is a necessity. However, these systems remain very difficult to use in practice primarily because they require both deep knowledge of formal methods and significant engineering effort.

Fiat2 is an ongoing research project aimed at addressing these challenges. It introduces a high-level Python-like programming language designed for data-intensive applications, providing developer productivity while also providing formal verification via the Coq proof assistant.

Data-intensive applications typically perform operations over structured datasets, such as filtering, joining, aggregating, and updating records. These computations are similar to the computations performed by relational database engines, where query planners are responsible for translating high-level declarative SQL queries into optimized execution plans. Some of the common optimizations used are predicate pushdown (applying a filter earlier to reduce the size of intermediate results), projection pushdown (similar to predicate pushdown), and join reordering.

While these optimizations are widely used in database systems, they are typically implemented without any formal correctness guarantees. Fiat2 extends the principles of rewrite-based optimizations to a general-purpose programming language, as opposed to query languages like SQL. Developers can write high-level programs with Python-like comprehensions over mutable table-like data structures, and Fiat2 will automatically optimize these programs using algebraic rewrites and internal data-structure selection.

By formally verifying each individual rewrite rule in Coq, Fiat2 ensures that all possible optimization schedules preserve the semantics of the original program. Also, unlike traditional database engines that have to optimize queries without prior knowledge of the specific workload, Fiat2 can leverage static analysis of the entire program to determine the best optimization choices for a given application, both in terms of the selected data structures and the schedule of rewrite rules to apply.

Chapter 2

Motivation & Related Work

Two significant predecessors to Fiat2 are Fiat and Castor. The original Fiat project explored the integration of relational query structures within a Coq-verified compilation framework and demonstrated verified compilation even down to assembly [1]. Fiat used deductive synthesis to refine programs by applying correctness-preserving transformations, known as refinement rules. A key feature of Fiat was its use of powerful automation tactics that could perform large-scale transformations on classes of programs. While these tactics allowed developers to synthesize correct implementations without having to apply each refinement rule manually, it was often hard to predict which kinds of programs these tactics would perform well on.

While Fiat achieved strong correctness guarantees, it suffered from usability and performance issues, mainly due to the fact that writing compiler scripts in Coq is difficult and also because Coq itself was not designed for high performance. As a result, it was limited to microbenchmarks rather than full-scale applications. Another drawback is that Fiat allows nondeterminism in the sense that it operates on mathematical sets and uses setoid rewrites in Coq instead of regular rewrites. Overall, Fiat2 builds on the foundational ideas of Fiat: it still uses Coq to verify transformations of relational queries but has a more standard verified-compiler approach with a deeply embedded syntax instead of relying on Coq scripts, uses deterministic semantics to avoid having to use setoid rewrites, and uses standard query optimizations rather than refinements.

Castor was a system that combined relational rewrite-based optimizations with automatic data-structure selection via a multistage intermediate representation, where the key idea was that rewrite rules can contain changes to both the data layout and the query logic, such that both aspects can be jointly optimized [2].

The main drawbacks of Castor were that its optimizer was heuristic-based and did not have any form of formal verification attached, and it only works for static databases (no updates to underlying data). Fiat2 builds on the foundational ideas of Castor by using Coq to verify each rewrite rule preserves the semantics and overall correctness, while also allowing dynamic applications. One other key change in Fiat2 compared to Castor is that, in order to determine which rewrite rules to apply on a given application, it will use an e-graph to store all the possible ways to rewrite the program soundly and then use equality saturation as well as a complicated cost model to pick the best rewrite plan [3].

Fiat2's approach of verifying individual rewrite rules in Coq to prove that a schedule of these individual rewrite rules preserves the semantics of the program is an approach that

has been successful in the past, as seen in multiple papers on tensor-kernel optimization, providing further evidence that this approach is practical and effective [4, 5]. Also, prior work such as HoTTSQL [6] has demonstrated the feasibility of formally verifying SQL rewrite rules in Coq, further supporting the practicality of verified query optimization.

Chapter 3

Overview of Verified Rewrites

In this section, we will cover the general ideas behind the rewrite-based query optimizations implemented in this thesis.

The next sections are organized as follows. Section 3.1 provides a brief overview of the Fiat2 language and its relevant constructs. Section 3.2 breaks down the main four categories of rewrites that we formally proved correct as part of this thesis. Section 3.3 then describes how we went from these rewrite categories to concrete theorem implementations in Coq. Finally, in Section 3.4, we walk through a small example program to demonstrate how these rewrites can be applied in practice to optimize a program.

3.1 Fiat2 Language Overview

Fiat2 is a Python-inspired statically typed high-level language that is designed to handle data-intensive applications where we would normally use a database engine like SQL. The language is built to support operating on objects that resemble relational data tables by, for example, looping over tables with comprehensions, filtering for certain records of the tables, projecting records out of tables, and joining two tables together.

To provide a concrete example, imagine we have a database storing information about books and authors that is split into two tables. Below are some examples of the types of queries that Fiat2 is designed to support.

```
// Filters
[ b.Title for b in Books if b.Genre == "Action" ]

// Projections
[ (b.Title, b.PublishDate) for b in Books ]

// Joins
[ (a.Name, a.Country, b.Title) for a in Authors for b in Books if a.Name == b.
    Author ]
```

As a side note, one key difference from Fiat2 and most other practical high-level languages is that all Fiat2 programs are guaranteed to terminate. In other words, we don't support arbitrary-length iteration via while loops for example, mainly because guaranteed termination greatly simplifies reasoning about the behavior of programs.

Internally, the parsed programs are represented as abstract syntax trees embedded within Coq, so the actual internal representation of the above examples will involve constructing trees with the following expressions.

```
Inductive expr : Type :=
| EVar (x : string)
| ELoc (x : string)
| EAtom (a : atom)
| EUnop (o : unop) (e : expr)
| EBinop (o : binop) (e1 e2: expr)
| EIf (e1 : expr) (e2 e3 : expr)
| ELet (e1 : expr) (x : string) (e2 : expr)
| EFlatmap (e1 : expr) (x : string) (e2 : expr)
| EFold (e1 e2 : expr) (v acc : string) (e3 : expr)
| ERecord (l : list (string * expr)) (* one row of db table *)
| EAccess (r : expr) (s : string) (* get record field / col value *)
| EDict (l : list (expr * expr))
| EInsert (d k v : expr)
| EDelete (d k : expr)
| ELookup (d k : expr)
| EOptMatch (e : expr) (e_none : expr) (x : string) (e_some : expr)
| EDictFold (d e0 : expr) (k v acc : string) (e : expr)
| ESort (l : expr)
(* Relational algebra expressions below *)
| EFilter (l : expr) (x : string) (p : expr)
(* Selects a subset of rows from table that satisfy a condition*)
| EProj (l : expr) (x : string) (r : expr)
(* Generalized projection of a row from table*).
| EJoin (l1 l2 : expr) (x y : string) (p r : expr)
(* Joins two tables where output rows satisfy a condition *)
```

Figure 3.1: Definition of All Fiat2 Expressions

The key expressions to look at are `ERecord`, `EAccess` for interacting with tables and then the relational algebra expressions `EFilter`, `EProj`, and `EJoin`. The first expression, `ERecord`, has a list of pairs of strings and expressions, representing a record of a table (a list of its field names and field values), corresponding to one row of a database. The second expression, `EAccess`, is used to retrieve the field value for a given field name and record, corresponding to getting a certain column's value in a row.

The next three expressions are the three main relational-algebra operators and are used in all of the theorems proven in this thesis.

The first one, `EFilter`, takes in a list of records representing a database table and a predicate expression, and it will output another list of records with all the records where the predicate returned true.

The second one, `EProj`, takes in a list of records representing a database table, and a transformation expression, and for each record, applies the transformation expression to it and returns a new list of the outputs.

The third one, `EJoin`, takes in two lists of records representing two database tables, a predicate expression, and a transformation expression. It first evaluates the predicate

expression on each pair of records in the input to determine what is included in the output, and then it evaluates the transformation expression on each pair of records.

Each of these three expressions also takes in one or more string arguments. Such a string represents the name of the variable that the expression expects the row of the input to be stored in. For example, when making an `EFilter` expression, in order for the predicate expression to use the value of the row it is checking, it needs to know what name it is bound to. To do this, this variable name is also passed to the `EFilter` expression so that it can set that variable's value to be each record in the table.

For reference, here is the code for interpreting these three expressions. Note that this does not represent how these expressions are actually implemented when compiled but rather just defines a deterministic way to compute what value is expected from an expression and lays out a specification for its behavior.

```
Fixpoint interp_expr (store env : locals) (e : expr) : value :=
    match e with
    ... (* all other cases omitted *)
    | EFilter l x p =>
        match interp_expr store env l with
        | VList l => VList (List.filter (fun v => interp_expr ... p) l)
        | _ => VUnit
        end
    | EProj l x r =>
        match interp_expr store env l with
        | VList l => VList (List.map (fun v => interp_expr ... r) l)
        | _ => VUnit
        end
    | EJoin l1 l2 x y p r =>
        match interp_expr store env l1 with
        | VList l1 =>
            match interp_expr store env l2 with
            | VList l2 => VList (flat_map
                    (fun v1 => List.map (fun v2 => interp_expr ... r)
                     (List.filter (fun v2 => interp_expr store ... p) l2)) l1)
            | _ => VUnit
            end
        | _ => VUnit
        end
    end.
```

Figure 3.2: Specification of Fiat2 Filter, Projection, and Join Behavior

In the `EFilter` and `EProj` cases, we first interpret the expression representing the table to get the list of records. Then, for `EFilter` we filter the list of records using the result of interpreting the predicate expression, and for `EProj` we map the list of records using the result of interpreting the transformation expression r.

For `EJoin`, we interpret the expressions representing the two tables we are joining to get two lists of records. We first filter out the pairs of records that don't satisfy the predicate expression (which takes in one record from each table), and then we map the remaining

records to the result of evaluating the transformation expression r with the pair of records from the input.

3.2 Types of Rewrites

The rewrite theorems we verify in this work fall in four main categories, which are all common rewrites used in query optimizations across both industry and academia [7, 8].

3.2.1 Filter + Join

This first category of theorems involve rearranging combinations of filters and joins. Specifically, the idea is that a filter of a join can be replaced with a join where the filter's logic is moved into the predicate expression of the join, essentially going from `(EFilter (EJoin . . . ))` to just `(EJoin . . . )`, which takes advantage of the fact that the joins in Fiat2 allow predicate expressions. Another possible rewrite is to add a filter on one of the input tables in the join if the join's predicate involves that table, going from `(EJoin . . . )` to `(EJoin (EFilter . . . ) . . . )`.

With these rewrites, filters that are on a join's output can become filters on the join's inputs. This is a common query optimization referred to as predicate pushdown, and the general idea behind it is that if the size of the input tables is reduced, then there will be less computation needed because the cross product needed for computing all the pairings for the join will be smaller.

3.2.2 Join + Projection

The next category of theorems involve rearranging combinations of joins and projections, and there are two variants that are implemented here. The first one involves replacing a projection of a join with just a join, where we modify the transformation expression of the join to include the projection, going from `(EProj (EJoin . . . ))` to just `(EJoin . . . )`, which takes advantage of the fact that joins in Fiat2 allow arbitrary expressions to compute each record of the join's output, not just a simple concatenation of the input columns.

The second variant involves adding a projection on one of the input tables of the join, where the projection we are adding removes all the columns of the table that are not needed in the join. The idea behind this optimization, much like the previous one, is that reducing the size of the input tables results in less required computation, except here we are reducing the size by making each record smaller rather than filtering out records.

3.2.3 Filter + Projection

The third category of theorems involve rearranging combinations of filters and projections, and there are two variants implemented here. The first involves adding a projection on the input table of the filter, where only the columns necessary for the filter are selected by the projection. The second involves commuting filters and projections, where an expression like `(EFilter (EProj . . . ))` can be replaced with `(EProj (EFilter . . . ))` assuming a few

conditions on the columns used in the filter and the projection, which will be discussed in more detail later.

3.2.4 Join Reordering

The last category of theorems involves rearranging joins with other joins, and there are again two variants that were explored. The first one involves commuting the two input tables of a join, showing that interpreting `(EJoin tb1 tb2)` and `(EJoin tb2 tb1)` output the same set of records. The second one covers associativity rather than commutativity, showing that interpreting `(EJoin (EJoin tb1 tb2) tb3)` and `(EJoin tb1 (EJoin tb2 tb3))` output the same set of records. Note that both of these differ from the previous three theorem categories in that they don't prove that the outputted lists of records are equivalent, only that they are permutations of each other. These join-reordering optimizations allow the order of nested joins to be arranged such that our intermediate results are as small as possible resulting in a more efficient execution plan overall.

3.3 Theorems

3.3.1 Filter + Join

The first theorem is written here, which as mentioned before, can replace a filter of a join with a new join where the join's predicate also handles the filter's predicate.

```
Theorem filter_into_join: forall (store env: locals) (Gstore Genv: tenv) (tb1
    tb2 p r pf: expr) (x y xf: string) (f1 f2: list (string * type)),
    (*assumptions omitted for now*) ->
    let pnew := EBinop OAnd p (ELet r xf pf) in
    interp_expr store env (EFilter (EJoin tb1 tb2 x y p r) xf pf)
    = interp_expr store env (EJoin tb1 tb2 x y pnew r).
```

Figure 3.3: `filter_into_join` Theorem Statement

We implement this by creating a new predicate called `pnew` that is the conjunction of the filter's predicate and the original join's predicate. Note that we often have to wrap variables in an `ELet` in order to rename bound variables such that they match their expected names in their new context. For example, we have to wrap `pf` in an `ELet` to set `xf` equal to `r`, because the original `pf` expression expects its input to be named `xf`.

The next two theorems are written below, which as mentioned before, can replace a join where one input table has a filter with a join where the join's predicate also handles the filter's predicate. Since this filter can be on the first or second input of the join, it is split into two theorems.

```
Theorem filter_into_join_left: forall (store env: locals) (Gstore Genv: tenv)
    (tb1 tb2 p r pf: expr) (x y xf: string) (f1 f2: list (string * type)),
     (*assumptions omitted for now*) ->
     let pnew := EBinop OAnd p (ELet (EVar x) xf pf) in
     interp_expr store env (EJoin (EFilter tb1 xf pf) tb2 x y p r)
     = interp_expr store env (EJoin tb1 tb2 x y pnew r).


Theorem filter_into_join_right: forall(store env: locals) (Gstore Genv: tenv)
    (tb1 tb2 p r pf: expr) (x y yf: string) (f1 f2: list (string * type)),
     (*assumptions omitted for now*) ->
     let pnew := EBinop OAnd p (ELet (EVar y) yf pf) in
     interp_expr store env (EJoin tb1 (EFilter tb2 yf pf) x y p r)
     = interp_expr store env (EJoin tb1 tb2 x y pnew r).
```

Figure 3.4: `filter_into_join_left` and `filter_into_join_right` Theorem Statements

Similar to the first theorem, we implement this by creating a new predicate `pnew` that is the conjunction of the filter's predicate and the original join's predicate. Note that if we had a filter on both tables of the join's input, we could use the left and right theorems above in sequence to rewrite both filters into the join predicate.

3.3.2 Join + Projection

The first theorem is written here, which as mentioned before, can replace a projection of a join with a new join where the join's transformation expression also handles the projection's expression.

```
Theorem proj_into_join: forall (store env: locals) (Gstore Genv: tenv) (t1 t2
    p r rp: expr) (x y xp: string),
    (*assumptions omitted for now*) ->
    let rnew := ELet r xp rp in
    interp_expr store env (EProj (EJoin t1 t2 x y p r) xp rp) =
    interp_expr store env (EJoin t1 t2 x y p rnew).
```

Figure 3.5: `proj_into_join` Theorem Statement

The implementation follows the same structure as the `filter_into_join` theorem above, where we construct a new transformation expression that composes the original join transformation with the projection's transformation.

The next two theorems are written below, which as mentioned before, can add a projection to either of the input tables of the join.

```
Theorem proj_pushdown_left: forall (store env: locals) (Gstore Genv: tenv) (
    tb1 tb2 p r: expr) (x y xp: string) (pcols rcols: list string) (f1 f2:
    list (string * type)) (t: type),
  (*assumptions omitted for now*) ->
  cols x p = Some pcols ->
  cols x r = Some rcols ->
  let columns := dedup String.eqb (pcols ++ rcols) in
  let rp := make_record xp columns in
  interp_expr store env (EJoin tb1 tb2 x y p r) =
  interp_expr store env (EJoin (EProj tb1 xp rp) tb2 x y p r).


Theorem proj_pushdown_right: forall (store env: locals) (Gstore Genv: tenv) (
    tb1 tb2 p r: expr) (x y xp: string) (pcols rcols: list string) (f1 f2:
    list (string * type)) (t: type),
  (*assumptions omitted for now*) ->
  cols y p = Some pcols ->
  cols y r = Some rcols ->
  let columns := dedup String.eqb (pcols ++ rcols) in
  let rp := make_record xp columns in
  interp_expr store env (EJoin tb1 tb2 x y p r) =
  interp_expr store env (EJoin tb1 (EProj tb2 xp rp) x y p r).
```

Figure 3.6: `proj_pushdown_left` and `proj_pushdown_right` Theorem Statements

The main difficulty in the implementation here was constructing the transformation expression `rp`, which needs to make a record of only the columns of the input table that were used in either the predicate or the transformation of the join.

This was broken into two parts. The first part was making a `cols` function that can be used to get back all the columns of a record that were used in a given expression. The second part was making a `make_record` function that, given a record, makes a new record with only a certain subset of the columns. These will be discussed in more detail later on.

3.3.3 Filter + Projection

The first theorem here is very similar to the second and third theorems in the above join + projection case and adds a projection to the input table of the filter. However, it is not correct to say that `(EFilter ...)` = `(EFilter (EProj ...) ...)` because adding the projection on the input table means that the records in the filter output will have less columns. To correct this, both the left-hand side and right-hand side are wrapped in a projection, where the columns that would have been missing due to the inner projection are not included by the outer projection.

```
Theorem proj_pushdown_filter: forall (store env: locals) (Gstore Genv: tenv) (
    tbl p r:expr) (x xi xp:string)
  (*assumptions omitted for now*) ->
  cols x p = Some pcols ->
  cols xp r = Some rcols ->
  let columns := dedup String.eqb (pcols ++ rcols) in
  let ri := make_record xi columns in
  interp_expr store env (EProj (EFilter tbl x p) xp r) =
  interp_expr store env (EProj (EFilter (EProj tbl xi ri) x p) xp r).
```

Figure 3.7: `proj_pushdown_filter` Theorem Statement

The next theorem, as mentioned before, involves commuting filters and projections. The main complicated part again was figuring out what columns the projection needed to have such that every expression would have all the columns it needs if the operators were commuted.

```
Theorem proj_filter_commute: forall (store env: locals) (Gstore Genv: tenv) (
    tbl p:expr) (x xp:string)
  (pcols rcols: list string) (f1: list (string * type)) (t: type),
  (*assumptions omitted for now*) ->
  cols x p = Some pcols ->
  incl pcols rcols ->
  let r := make_record xp rcols in
  cols xp r = Some rcols ->
  interp_expr store env (EProj (EFilter tbl x p) xp r) =
  interp_expr store env (EFilter (EProj tbl xp r) x p).
```

Figure 3.8: `proj_filter_commute` Theorem Statement

One thing to note here is that the left-hand side transformation expression must be of the form that is returned by `make_record` in order to use this theorem. Also, the columns used in the predicate expression (`pcols`) must be a subset of the columns used in the transformation expression (`rcols`) so that if the projection happened first, the filter would still have all of the columns it needs to compute its predicate result for each record.

3.3.4 Join Reordering

These last two theorems are structured differently from the rest, in that they are proving that the two lists outputted from interpreting the two expressions are permutations of each other, meaning that they contain the same elements but not necessarily in the same order.

```
Theorem join_comm: forall (store env: locals) (Gstore Genv: tenv) (tb1 tb2 p r
    : expr) (x y: string) (l1 l2: list value),
    interp_expr store env (EJoin tb1 tb2 x y p r) = VList l1 ->
    interp_expr store env (EJoin tb2 tb1 y x p r) = VList l2 ->
    Permutation.Permutation l1 l2.
```

Figure 3.9: `join_comm` Theorem Statement

The first theorem above is fairly straightforward and simply states that the input tables to a join can be commuted. The second theorem below, covering join associativity, is much more complicated and required a lot of careful thought to formally state correctly.

```
Theorem join_assoc: forall (store env: locals) (Gstore Genv: tenv) (tb1 tb2
    tb3 p1 p12 r12: expr) (x y z xy yz:string) (l1 l2: list value) (xcols
    ycols zcols: list string),
     let rx := make_record x xcols in
     let ry := make_record y ycols in
     let rz := make_record z zcols in
     let rxy := ERecord (("0"%string, rx) :: ("1"%string, ry) :: nil) in
     let ryz := ERecord (("0"%string, ry) :: ("1"%string, rz) :: nil) in
     let y_yz := EAccess (EVar yz) "0" in
     let z_yz := EAccess (EVar yz) "1" in
     let p12let := (ELet z_yz z (ELet rxy xy p12)) in
     let p23 := EBinop OAnd p1let p12let in
     let r23 := (ELet z_yz z (ELet (EBinop OConcat (EVar x) y_yz) xy r12)) in
     interp_expr store env (EJoin (EJoin tb1 tb2 x y p1 rxy) tb3 xy z p12 r12)
    = VList l1 ->
     interp_expr store env (EJoin tb1 (EJoin tb2 tb3 y z (EAtom (ABool true))
    ryz) x yz p23 r23) = VList l2 ->
     Permutation.Permutation l1 l2.
```

Figure 3.10: `join_assoc` Theorem Statement

Some of the complications included having to move the predicate logic to only the outer join on the right side by making the inner join's predicate just the value `true`, having to construct records that combine two of the tables for the inputs to the outer joins, etc.

3.4 Example Derivation

To illustrate how the rewrite theorems from this thesis can be applied in practice, consider a simple example involving two tables called `Books` and `Authors`, where `Books` has the field names `[Title, Author, PublishDate, Publisher, Genre]`, and `Authors` has the field names `[Name, Country]`. Imagine we want to perform a query like the following:

```
[ (b.Title, a.Country)
  for b in Books
  for a in Authors
  if b.Author == a.Name ]
```

Before any optimizations occur, this would be internally represented in Fiat2 as an expression that looks like the following, where we are performing a join of the two tables, where the predicate is `b.Author == a.Name`, and the transformation is a record with `b.Title` and `a.Country`:

```
(EJoin Books Authors "b" "a"
   (EBinop OEq (EAccess (EVar "b") "Author")
               (EAccess (EVar "a") "Name"))
   (ERecord [("Title", EAccess (EVar "b") "Title");
```

```
            ("Country", EAccess (EVar "a") "Country")]))
```

One inefficiency in the above expression is that we are performing a join of all five fields in the `Books` table even though we only need the title and author, resulting in an unnecessarily large join output, which will slow down our program. Luckily, with our `proj_pushdown_left` theorem, we can rewrite this expression to add a projection to the books table inside the join, such that we only project the needed columns before executing the join. This would result in a revised expression that looks like the following, where we add a projection to the first table in the join.

```
(EJoin
  (EProj books "b1"
    (ERecord [("Author", EAccess (EVar "b1") "Author");
              ("Title", EAccess (EVar "b1") "Title")]))
  Authors
  "b" "a"
  (EBinop OEq (EAccess (EVar "b") "Author")
                (EAccess (EVar "a") "Name"))
  (ERecord [("Title", EAccess (EVar "b") "Title");
            ("Country", EAccess (EVar "a") "Country")]))
```

This rewrite allows for a much more performant program because evaluating the join is now less computationally expensive since the input table has smaller records than before. It is also proven to preserve the semantics of the program, demonstrating the effectiveness of proving these rewrite theorems as a means of achieving verifiably correct database optimizations.

Chapter 4

Proof Strategy

In this section, we will first discuss more carefully what exactly the theorems are stating and what notion of equality is being used. Then, we will go over some common proof assumptions that show up in most / all of the theorems. Finally, we will give a brief overview of Coq and demonstrate how it is used to prove theorems.

4.1 Semantic Preservation

Before we dive into the proofs, let's first define more specifically what these theorem statements are stating exactly. Most of the theorem statements for these query-optimization rewrites are of the form `interp_expr exprA = interp_expr exprB`, where these expressions `exprA` and `exprB` are some combination of relational-algebra operators on some input tables. This is stating that the list of records we get back from interpreting `exprA` should be the same exact list of records that we get back from interpreting `exprB`. Specifically, this requires the order of the two lists to be the same, so if the left side results in `[recordA, recordB]`, and the right side results in `[recordB, recordA]`, these are not considered equivalent.

This idea of semantic preservation that we are using here is called list semantics. A lot of other related works that involve similar query optimizations use a different type of semantic preservation referred to as set semantics, meaning that we just care about the set of records returned and don't require the records to be in the same order.

We prefer list semantics over set semantics for several reasons. One reason is that set semantics would introduce nondeterminism into our system, meaning that the order that records are read out of a table can vary. This can lead to security vulnerabilities, because attackers could conclude private information about other people's data usage based on what order records are read in. Another more practical reason is that it is much easier to prove the theorems mentioned above when the order of records is fixed, both due to being easier to reason about conceptually and being easier to implement in Coq.

One difficulty that arose due to this choice is that we have to change the join-reordering theorems such that they prove the `interp_expr` result on the left and the `interp_expr` result on the right are permutations of each other, but not that they are equivalent. This makes it possible to prove the theorems but more complicated to apply because we can't use a standard rewrite like with all the other theorem statements.

4.2 Common Proof Assumptions

Next, we will go over the key assumptions for each of these theorems, which are the conditions that have to be met to apply the rewrite in the theorem statement. Here is a quick overview of the common assumptions that show up in most of these theorems.

```
tenv_wf Gstore -> tenv_wf Genv ->
locals_wf Gstore store -> locals_wf Genv env ->
```

These four assumptions appear in almost all the theorems, and they say that the environments that contain all the values of variables (`store` and `env`) and the type environments that contain all the types of variables (`Gstore` and `Genv`) are well-formed. Specifically, the first two `tenv_wf` assumptions say that every type inside the type environment is well-formed, and the last two `locals_wf` assumptions say that for every variable in the environment, the type of the variable's value is correctly mapped in the type environment.

```
type_of Gstore Genv expr t ->
```

Most theorems will also have a few `type_of` assumptions, where the assumption `type_of Gstore Genv expr t` means that when evaluating the expression `expr` in the given `Gstore` and `Genv`, the result has the type `t`. For example, saying `type_of Gstore Genv tb1 (TList (TRecord f1))` means that `tb1` evaluates to a list of records.

```
free_immut_in var expr = false ->
```

The assumption above says that the expression `expr` does not use the variable `var`. The main reason we would need this kind of assumption is for cases where we are evaluating an expression in an environment with more bindings than it originally had, and we want to make sure that these extra assignments do not affect the result.

```
cols var expr = Some expr_cols ->
let new_expr := make_record var expr_cols in
```

This is one of the more complicated and powerful assumptions; it says that out of all the field names of the record stored in `var`, the expression `expr` only uses `expr_cols` of them, where `expr_cols` is a list of the field names. It is often followed by a call to make_record which will reconstruct the record with only the `expr_cols` columns.

This assumption is a key for optimizing queries because it can be used to remove irrelevant columns of expressions from the projection, reducing the size of the input tables in complex expressions.

4.3 Coq Overview

Coq is a proof assistant and functional programming language that is used to develop formal mathematical proofs that can be checked automatically, and in this thesis it is used to both define the Fiat2 language and to prove all the rewrite theorems covered above.

A theorem written in Coq takes a proposition, for example the statement that Fiat2 expressions evaluate to the same result, and can have any number of assumptions / preconditions that are required for the proposition to be true. To illustrate the process of proving a theorem, we will break down a simple unrelated theorem that states that adding 0 to any number gives back the original number:

```
Lemma plus_0_r : forall n : nat, n + 0 = n.
  induction n as [| n' IH].
  - simpl. reflexivity.
  - simpl. rewrite IH. reflexivity.
Qed.
```

The general process of proving a theorem in Coq involves making incremental changes to both the current goal and the current hypotheses until the goal becomes trivial. The goal starts out as the conclusion of the theorem, and the hypotheses start out as all the assumptions of the theorem statement.

The way these incremental changes are made is through the use of tactics. For example, the `induction` tactic takes in a variable and splits the goal into subgoals representing the base cases and the inductive cases with respect to that variable, and the `rewrite` tactic takes in another lemma / theorem of the form `x = y` and tries to replace occurrences of `x` with `y` in the goal.

Chapter 5

Proofs

In this section, we will first do a detailed walkthrough of one proof, specifically the proof of the `filter_into_join_left` theorem covered above, to give an example of the general mechanics and strategy of proving these relational-algebra theorems. Then, we will zoom out and discuss the challenging parts of all of the proofs from all the different categories mentioned above.

5.1 Filter + Join Theorem Proof Deep Dive

We will first go over the full proof of the `filter_into_join_left` theorem, which can be found here and is also copied into the appendix section.

Before diving into the actual proof, let's first look at the assumptions of the proof in more detail. The first four are the standard assumptions that state the environments and type environments are well-formed. The next few `type_of` assumptions are saying that certain expressions have certain types. Specifically, the first one says the predicate expression `p` must be of type `TBool`. Since the expression is only interpreted after `x` and `y` are set to the records from the two tables in the environment, the `type_of` assumption must also add the types of `x` and `y` into the type environment. The other three `type_of` assumptions are similar.

Next, there are the two `free_immut_in . . . = false` assumptions. As a reminder, the first one says that the expression `pf` never uses the variable `x`. The reason this assumption is needed is because in the left-hand side expression, `pf` only has `xf` set in its environment, and in the right-hand-side expression, `x` and `y` are also set in its environment. If the expression `pf` were to use `x` and `y`, then it would be possible for `pf` to evaluate differently on the left-hand side and the right-hand side. The final assumption simply says `x` cannot be equal to `y` because if they were, then one of them would be overwritten by the other in the environment.

Moving back to the actual proof, after instantiating all the variables and simplifying, the goal starts off looking like the following, where we have expressions on the left side and on the right side that we have to show are equal.

The general strategy for almost all the proofs is to start by breaking down the outermost expressions on both sides, which in this case are the two match statements that get the list of records from each input table. These are broken down by deconstructing them and performing case analysis on the interpreted outputs and then using the `type_of` assumptions

```
match
  match interp_expr store env tb1 with
  | VList l => VList (filter ... l)
  | _ => VUnit
  end
  with
  | VList l1 =>
    match interp_expr store env tb2 with
    | VList l2 => VList (flat_map (fun v1 : value => map ... (filter ... l2))
   l1)
    | _ => VUnit
    end
  | _ => VUnit
  end =
  match interp_expr store env tb1 with
  | VList l1 =>
    match interp_expr store env tb2 with
    | VList l2 => VList (flat_map (fun v1 : value =>
                  map ... (filter (fun v2 : value => apply_bool_binop andb ...
   ... ) l2)) l1)
    | _ => VUnit
    end
  | _ => VUnit
end
```

Figure 5.1: Initial Goal State of `filter_into_join_left` Proof

to narrow the cases down to just the one where the output is a list of records. After that, the goal looks roughly like the following, where `b` is the result of evaluating `pf` with the `type_of` assumption.

From this point, the `interp_expr` of `p` is also deconstructed into a boolean with its `type_of` assumption again, and then there are two goals generated (one for each value the boolean could be). For both goals, the final steps of the proof boil down to showing the original predicate and the newly constructed predicate have the same result. This mainly required reordering the `map.put` and `map.get` operations that are used to interact with the environment so that the left-hand side better matches the right-hand side, as well as using the `free_immut_in . . . = false` assumptions to show that a binding can be removed from the environment without changing the result of interpreting the expression.

5.2 Proof Challenges

Next, here are some of the biggest challenges that came up while writing the proofs for all the theorems mentioned above, split into each proof category.

```
(if b then
map (fun v2 : value => interp_expr store (map.put (map.put env x a) y v2) r)
  (filter
    (fun v2 : value => interp_expr store (map.put (map.put env x a) y v2) p)
  l0)
 else nil) =
map (fun v2 : value => interp_expr store (map.put (map.put env x a) y v2) r)
(filter
  (fun v2 : value =>
      apply_bool_binop andb (interp_expr store (map.put (map.put env x a) y
 v2) p)
        (interp_expr store
          (map.put (map.put (map.put env x a) y v2) xf
              map.get (map.put (map.put env x a) y v2) x) pf)) l0)
```

Figure 5.2: Intermediate Goal State of `filter_into_join_left` Proof

5.2.1 Filter + Join

For the filter + join proofs, there were no major challenges. The `filter_into_join_left` theorem was covered in detail above, and the other two theorems were proven in a very similar way with essentially the same strategy.

5.2.2 Join + Projection

The join and projection theorems were where most of the complexity arose across all the theorems. The first theorem, `proj_into_join`, was very straightforward as it simply rewrote a projection of a join to a join where the projection's transformation is composed with the join's transformation. The next two theorems, `proj_pushdown_left` and `proj_pushdown_right`, were much trickier to define and required a lot more careful thought and overall lines of code.

Below is the full theorem statement for `proj_pushdown_left`.

As a reminder, this theorem states that we can add a projection to one of the inner tables of a join, as long as a few conditions are met. Specifically, this theorem makes the projection simply return a new record that only contains the column names that are used in either the predicate or transformation expressions of the joins.

The core of the complications starts with the `cols` function, which is implemented with a fixpoint algorithm that matches on the type of expression. Ideally, this function would just return a list of the column names; however, it instead returns an option type (either `Some list` or `None`). We chose this so that we can also represent the case where it isn't known which particular columns of a record an expression uses. For example, when computing `(cols x (EVar x))`, the expression could use any of the columns because it has access to the whole record, so `None` is returned to represent that.

Moving on in the actual proof of the theorem, it starts off similar to our other theorems. Eventually, we reach a goal that looks like this, where `a` is one of the records in the first input table, and `a'` is the new record that only has the necessary columns to compute the join, and we need to show that interpreting the expression `p` gives the same result whether `a` or `a'` is used.

```
Theorem proj_pushdown_left: forall (store env: locals) (Gstore Genv: tenv) (
    tb1 tb2 p r: expr) (x y xp: string) (pcols rcols: list string) (f1 f2:
    list (string * type)) (t: type),
  tenv_wf Gstore -> tenv_wf Genv ->
  locals_wf Gstore store -> locals_wf Genv env ->
  type_of Gstore (map.put (map.put Genv x (TRecord f1)) y (TRecord f2)) p
    TBool ->
  type_of Gstore (map.put (map.put Genv x (TRecord f1)) y (TRecord f2)) r t ->
  type_of Gstore Genv tb1 (TList (TRecord f1)) ->
  x <> y ->
  cols x p = Some pcols ->
  cols x r = Some rcols ->
  let columns := dedup String.eqb (pcols ++ rcols) in
  let rp := make_record xp columns in
  interp_expr store env (EJoin tb1 tb2 x y p r) =
  interp_expr store env (EJoin (EProj tb1 xp rp) tb2 x y p r).
```

Figure 5.3: `proj_pushdown_left` Full Theorem Statement

```
  match interp_expr store (map.put (map.put env y a0) x a) p with
  | VBool b => b
  | _ => false
  end =
  match interp_expr store (map.put (map.put env y a0) x a') p with
  | VBool b => b
  | _ => false
  end
```

This is a key idea in the proof and is needed at multiple points in the proof and in other proofs, so it was defined as a separate lemma, which is written below.

```
Lemma rel_lemma: forall (store: locals) (x: string) (a a': value) (e: expr) (
    columns: list string) tl tl',
  forall (env: locals),
  type_of_value a (TRecord tl) ->
  type_of_value a' (TRecord tl') ->
  cols x e = Some columns ->
  relation a a' columns ->
  interp_expr store (map.put env x a) e
  = interp_expr store (map.put env x a') e.
```

Note that `relation a a' columns` means that `a'` is a subset of `a`, and `columns` is a subset of both `a` and `a'`. Given that, this lemma says the following: if `a` is a record with columns `tl`, `a'` is a record with columns `tl'`, and the expression `e` only uses the columns `columns` out of `x`, then if the aforementioned relation holds, interpreting the expression `e` gives the same result whether using `a` or `a'`.

The proof of this lemma was long and difficult, requiring the use of structural induction on the expression `e`, meaning that for each expression case, there is an induction hypothesis generated for each of the sub expression(s). Another tricky part was to prove the `type_of_value a' (TRecord tl')` assumption in order to actually use this lemma, which was a fairly complex proof in itself that was proven in another separate lemma called `record_type_a'`.

One of the assumptions for the above `record_type_a'` lemma was that the columns we got from the `cols` function are a subset of the columns of the input table. To satisfy this assumption, the `cols_in_record` lemma was defined which is included below.

```
Lemma cols_in_record: forall (Gstore Genv: tenv)(e: expr)(t: type)(x: string)(
    f1: list (string*type))(ecols: list string),
    type_of Gstore Genv e t ->
    map.get Genv x = Some (TRecord f1) ->
    cols x e = Some ecols ->
    incl ecols (map fst f1).
```

This lemma is essentially confirming that the definition of the `cols` function is accurate in the sense that the list of columns it returns should always be included in the list of all the columns in the record. This also required a fairly long proof involving induction on the `type_of` assumption which goes through each possible type of the expression.

Through the use of all of these lemmas, the rest of the proof of the actual theorem was fairly straightforward.

5.2.3 Filter + Projection

The two filter + projection theorems mostly followed the structure of the join + projection theorems, and used many if not all of the same lemmas discussed above.

5.2.4 Join Reordering

For the join-reordering theorems, the main challenge was proving that the results of interpreting the two expressions were permutations of each other instead of equivalent to each other. This was done by making and proving a lemma that allows flatmaps to be commuted and then rewriting using that, which makes use of Coq's setoid rewrites.

Using the above lemma, the `join_comm` theorem had a fairly short proof. For the `join_assoc` theorem which proves the associativity of joins, the proof for that lemma currently isn't finished yet, but it is expected to follow the same logic as the join commutativity theorem.

Chapter 6

Conclusion

Throughout this thesis, we have both correctly formulated several relational-algebra query rewrite theorems in Fiat2 and formally proved that these theorems are correct. We focused on four categories of rewrites, with the first involving rearranging filters and joins, the second involving rearranging joins and projections, the third involving rearranging filters and projections, and the fourth involving rearranging joins with other joins. For each of these categories, we proved multiple rewrite theorems are correct, enabling several common optimizations in database literature to be applied to Fiat2 programs with full confidence that the semantics of the program are preserved. These optimizations include join reordering, predicate pushdown, and projection pushdown among others.

Proving each theorem was nontrivial and required careful consideration of how to break down the proof into reusable lemmas, how to keep track of which columns of records are used and prove that removing certain columns from a record won't change the result, how to prove two expressions output the same set of records, etc.

This work lays the foundation for integrating verified query optimization into Fiat2, enabling people to write high-level programs for data-intensive applications in a user-friendly syntax and have their queries automatically optimized under the hood with proven correctness guarantees for each optimization.

Chapter 7

Future Work

While we have made significant progress with all the theorems discussed so far, Fiat2 is still in its early stages of development, and there are several areas to explore in the future. The first priority would be to finish up the join associativity proof. After that, all the theorems will be completely proven.

Of course, the reason these theorems were proven in the first place was so that they could be used to optimize programs. So the next step is to construct an optimizer that can look at a program and determine which rewrites to do. The plan for this, which is currently being worked on, is to use an e-graph engine that will have a model of measuring the "cost" of programs and find an optimal schedule of rewrites to apply based on that model.

So far, our approach for optimizing programs has been via rewriting relational algebra expressions. But, there are several other ways to optimize programs that are frequently used in database systems that can be implemented in Fiat2. One such example is incorporating index structures into Fiat2, which is actively being worked on by another researcher in the group. This would require adding support for index transformations and proving that the transformations are sound.

Similarly, another avenue that is currently being explored is supporting and optimizing aggregate queries by updating additional metadata after every query and proving that this metadata is always accurate. For example, to support a query that finds the record with the maximum value of a certain column, one way to implement this would be to always store what the maximum value is, and whenever a record is inserted or modified, update this maximum value if the new value is bigger.

Another area to consider is the actual implementation of the joins. For reference, there are several common ways to implement a join of two tables, and each one has its own performance characteristics and situations where they are more appropriate than others. These include nested loop joins, hash joins, merge joins, etc. Perhaps we could also implement some form of program analysis that will decide on a join implementation based on the nature of the program and which of the join implementations would perform best.

Appendix A

`filter_into_join_left` Theorem Full Proof

```
Theorem filter_into_join_left: forall (store env: locals) (Gstore Genv: tenv)
    (tb1 tb2 p r pf: expr) (x y xf: string) (f1 f2: list (string * type)),
     tenv_wf Gstore -> tenv_wf Genv -> locals_wf Gstore store -> locals_wf Genv
     env ->
     type_of Gstore (map.put (map.put Genv x (TRecord f1)) y (TRecord f2)) p
    TBool ->
     type_of Gstore (map.put Genv xf (TRecord f1)) pf TBool ->
     type_of Gstore Genv tb1 (TList (TRecord f1)) ->
     type_of Gstore Genv tb2 (TList (TRecord f2)) ->
     free_immut_in x pf = false ->
     free_immut_in y pf = false ->
     x <> y ->
     let pnew := EBinop OAnd p (ELet (EVar x) xf pf) in
     interp_expr store env (EJoin (EFilter tb1 xf pf) tb2 x y p r)
     = interp_expr store env (EJoin tb1 tb2 x y pnew r).
Proof.
intros store env Gstore Genv tb1 tb2 p r pf x y xf f1 f2 WF1 WF2 L1 L2 TP TPF
    T1 T2 FX FY XY. simpl.
assert (TW1: type_wf (TRecord f1)). 1: { apply type_of__type_wf in T1; auto.
    inversion T1; auto. }
assert (TW2: type_wf (TRecord f2)). 1: { apply type_of__type_wf in T2; auto.
    inversion T2; auto. }
destruct (interp_expr store env tb1) eqn:H1; auto. destruct (interp_expr store
     env tb2) eqn:H2; auto. f_equal. rewrite flat_map_filter.
apply In_flat_map_ext. intros a LA. eapply type_sound in T1; eauto. inversion
    T1. rewrite H1 in H. injection H as H. subst l1 t.
apply Forall_In with (x:=a) in H3; auto. eapply type_sound with (store:=store)
     (env:=map.put env xf a) in TPF; eauto.
3: { apply locals_wf_step; auto. } 2: { apply tenv_wf_step; auto. } inversion
    TPF. unfold get_local. destruct b.
- f_equal. apply In_filter_ext. intros b LB. rewrite map.get_put_diff; auto.
    rewrite map.get_put_same.
     eapply type_sound in T2; eauto. inversion T2. rewrite H2 in H. injection H
     as H. subst l1 t. apply Forall_In with (x:=b) in H5; auto.
     eapply type_sound with (store:=store) (env:=map.put (map.put env x a) y b)
     in TP; eauto.
```

```
    3: {apply locals_wf_step; auto. apply locals_wf_step; auto. } 2: {apply
    tenv_wf_step; auto. apply tenv_wf_step; auto. }
    inversion TP. simpl. destruct (string_dec y xf).
    + subst xf. rewrite Properties.map.put_put_same. rewrite Properties.map.
    put_put_diff; auto.
    rewrite <- not_free_immut_put_sem with (x:=x)(v:=a); auto. rewrite <- H0.
    rewrite Bool.andb_true_r. auto.
    + rewrite Properties.map.put_put_diff; auto. destruct (string_dec x xf).
    * subst xf. rewrite Properties.map.put_put_same. rewrite <-
    not_free_immut_put_sem with (x:=y)(v:=b); auto. rewrite <- H0.
        rewrite Bool.andb_true_r. auto.
    * rewrite Properties.map.put_put_diff with (k2:=xf); auto. rewrite <-
    not_free_immut_put_sem with (x:=y)(v:=b); auto.
        rewrite <- not_free_immut_put_sem with (x:=x)(v:=a); auto. rewrite <-
    H0. rewrite Bool.andb_true_r. auto.
- symmetry. apply map_nil. apply filter_nil. intros b LB.
    rewrite map.get_put_diff; auto. rewrite map.get_put_same. destruct (
    string_dec y xf).
    + subst xf. rewrite Properties.map.put_put_same. rewrite Properties.map.
    put_put_diff with (k2:=y)(v2:=a); auto.
    rewrite <- not_free_immut_put_sem with (x:=x)(v:=a); auto. rewrite <- H0.
    unfold apply_bool_binop.
    destruct (interp_expr store (map.put (map.put env x a) y b) p); auto.
    rewrite Bool.andb_false_r. auto.
    + rewrite Properties.map.put_put_diff with (k2:=xf); auto. destruct (
    string_dec x xf).
    * subst xf. rewrite Properties.map.put_put_same. rewrite
    not_free_immut_put_sem with (x:=y)(v:=b) in H0; auto. rewrite <- H0.
        unfold apply_bool_binop. destruct (interp_expr store (map.put (map.put
     env x a) y b) p); auto. rewrite Bool.andb_false_r. auto.
    * rewrite Properties.map.put_put_diff with (k2:=xf); auto. rewrite
    not_free_immut_put_sem with (x:=x)(v:=a) in H0; auto.
        rewrite not_free_immut_put_sem with (x:=y)(v:=b) in H0; auto. rewrite
    <- H0. unfold apply_bool_binop.
        destruct (interp_expr store (map.put (map.put env x a) y b) p); auto.
    rewrite Bool.andb_false_r. auto.
Qed.
```

Appendix B

GitHub Link

The full source code can be found at:
github.com/mit-plv/fiat2/.../Optimize.v

References

[1] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. "Fiat: Deductive synthesis of abstract data types in a proof assistant". In: *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2015. URL: http://adam.chlipala.net/papers/FiatPOPL15/.

[2] J. K. Feser, S. Madden, N. Tang, and A. Solar-Lezama. "Deductive optimization of relational data storage". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 170:1–170:30. DOI: 10.1145/3428238. URL: https://doi.org/10.1145/3428238.

[3] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. "Egg: Fast and extensible equality saturation". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. URL: https://dl.acm.org/doi/pdf/10.1145/3434304.

[4] A. Liu, G. L. Bernstein, A. Chlipala, and J. Ragan-Kelley. "Verified tensor-program optimization via high-level scheduling rewrites". In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498717. URL: https://doi.org/10.1145/3498717.

[5] A. Liu, G. Bernstein, A. Chlipala, and J. Ragan-Kelley. "A verified compiler for a functional tensor language". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656390. URL: https://doi.org/10.1145/3656390.

[6] S. Chu, K. Weitz, A. Cheung, and D. Suciu. "HoTTSQL: Proving query rewrites with univalent SQL semantics". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2017, pp. 510–524. ISBN: 9781450349888. DOI: 10.1145/3062341.3062348. URL: https://doi.org/10.1145/3062341.3062348.

[7] S. Chaudhuri. "An Overview of Query Optimization in Relational Systems". In: *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 1998, pp. 34–43. DOI: 10.1145/275487.275492. URL: https://doi.org/10.1145/275487.275492.

[8] M. Jarke and J. Koch. "Query Optimization in Database Systems". In: *ACM Comput. Surv.* 16.2 (1984), pp. 111–152. DOI: 10.1145/356924.356928. URL: https://doi.org/10.1145/356924.356928.