Formalisation of Relational Algebra and a SQL-like Language with the RISCAL Model Checker

Joachim Borya

May 2023

RISC Report Series No. 23-06

ISSN: 2791-4267 (online)

Available at `https://doi.org/10.35011/risc.23-06`

JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Joachim Borya

Submitted at
RISC
Research Institute for
Symbolic Computation

Supervisor
A. Univ.-Prof. DI Dr.
Wolfgang Schreiner

May 2023

Formalisation of Relational Algebra and a SQL-like Language with the RISCAL Model Checker

Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Technische Mathematik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Zusammenfassung

Das relationale Datenbankmodell beruht auf dem mathematischen Konzept der Relationenalgebra. Um Daten schnell verfügbar zu machen, ohne eigens Prozeduren für den Zugriff zu erstellen, die von der internen Repräsentation der Daten abhängig sind, wurden Abfragesprachen entwickelt. SQL (structured query language) kann als Quasistandard hierfür gesehen werden. In dieser Arbeit geht es um die Formalisierung und Verifikation der Relationenalgebra und einem kleinen, aber elementaren Teilmenge von SQL mit Hilfe des RISCAL Model Checkers, eines Software-Werkzeuges zur formalen Spezifikation und Verifikation von mathematischen Theorien und Algorithmen.

Abstract

The relational database model is based on the mathematical concept of relational algebra. Query languages have been developed to make data available quickly without creating dedicated access procedures that depend on the internal representation of the data. SQL (structured query language) can be seen as a quasi-standard for this. This thesis deals with the formalization and verification of relational algebra and a small but elementary subset of SQL with the help of the RISCAL model checker, a software tool for the formal specification and verification of mathematical theories and algorithms.

Contents

1. Introduction

1.1. Background

Relational databases are used in many different ways in software products. According to their name, they are based on the mathematical concept of the relation, which can be represented as a table. Each column has a fixed meaning as an attribute with a specific data type, which in turn describes a component of an object represented by the row (also tuple) [1]. A real-world example would be a list of students with their contact details, matriculation numbers, and courses. A database is a collection of such tables.

Different database languages have been devised to define, manipulate, and query data in the past [2]. Relational algebra (RA) is the theory underlying relational database languages. It is used for theoretical considerations and is already fully described by a few operators on relations (functions in the mathematical sense) [3]–[5]. SQL (Structured Query Language) is a standardized, relational database language that is used, among other things, to query data. The syntactic construct that makes this possible is the SPJ expression (Select-Project-Join) [6].

The formal description of such a language consists of its syntax in the sense of correctly formed expressions and the meaning of these expressions, i.e., their semantics [7]. The syntax of a language can be represented by EBNF (Extended Backus Naur Form) or a syntax diagram. Within the framework of denotational semantics, each valid expression is assigned an element from a previously defined semantic domain via structural induction, which makes it clear what exactly the interpretation of this expression is [7].

RA can be considered a logical theory, dividing its language into symbols for constants, functions, and predicates for which certain axioms are supposed to hold [8]. A model of the theory is an interpretation of those symbols so that every formula becomes a true statement for all variable assignments. There may be an infinite number of such models [7]. It is well known that first-order logic is undecidable. However, if one restricts oneself to domains of finite size, some procedures can decide whether a formula is valid in the theory [7]. This is the task of so-called model checkers, whose application areas are often software products and less often mathematical theories [7]. If a finite instance is found that the theory does not model, a proof for all possible domains becomes superfluous.

The RISC Algorithmic Language (RISCAL) is an example of such a model checker [9], [10]. It uses a language based on mathematical notation and can be used to specify mathematical theories and algorithms. In addition, it is possible to automatically compile RISCAL into the LISP-like language SMT-LIB [11] to delegate proof problems to external SMT (Satisfiability Modulo Theories) solvers such as Z3 [12]. Recently RISCAL has been extended via the RISCTP theorem proving interface to include external theorem provers by which models of arbitrary sizes can be verified [13], [14].

RISCAL has been developed as a tool to simplify the verification of algorithms. The verification of an algorithm with respect to a formal specification, roughly speaking, rests on three pillars. While preconditions provide information about the allowed inputs, postconditions deal with the correctness of the result. Finally, loop invariants provide a way to verify relationships and value ranges of variables in each iteration of the loop [7].

1.2. Goals and Results

1.2.1. Goals

The thesis aims to formalize the relational algebra and a prototypical query language with RISCAL. This has the purpose that we can validate the correctness of all operations and algorithms on finite domains. This will increase the reader's trust that the definitions do what they should. We will also demonstrate that the query language is as expressive as a subset of SQL.

1.2.2. Achieved Results

In the case of RA, we defined a mathematical theory (section 3.1), fed it into RISCAL (section 3.2), and tested it with the means at its disposal (section 3.3). An attempt is also made here to formulate suitable and efficient algorithms for carrying out the operations described and to subject them to validation and benchmarking.

The part for SQL is similar, except that it inherits the underpinning from the RA and is manifested through the formalization of an abstract syntax (section 4.1) and denotational semantics (section 4.2) of SQL. The syntax and semantics are then implemented in RISCAL (section 4.3) in order to carry out sample queries afterward (section 4.4).

1.3. Structure of the Thesis

To begin the thesis, we create an overview of databases and their theory, program verification, and the basics of formal languages in Chapter 2. The essential content of the thesis consists of a part on RA (Chapter 3) and one on SQL (Chapter 4), the second building on the first.

Near the end, we will summarize the key findings of the thesis (Chapter 5). In succession to the thesis, we will provide a full version of the source code (Appendix A) and details about the system architecture on that the test was run (Appendix B).

2. State of the Art

This chapter provides an overview of the main topics covered in this thesis, including the relational model for databases, formal methods for program and system verification, and the foundations of formal languages in language design. By exploring their history and fundamental concepts, we can better understand the motivation behind the work presented in this thesis.

2.1. Relational Databases

History

Nowadays, data are generated in almost every area, and software systems depend on them to function properly. For database users, which can be programs or people, it is practical if they do not have to deal with the concrete appearance of the data in memory because, if these changes, all previously implemented access routines are useless in the worst case. (Logical) data independence [1] is needed, i.e., instead of parsing the data using a procedural programming language, in modern database management systems (DBMS), it is sufficient to specify which data should be queried; this is called the *declarative paradigm* [15]. An example of such a query would be something like the following:

> Every seat S that is reserved in train Z at time T.

Currently, the best-known language to query data is the Structured Query Language (SQL), which is based on a two-dimensional, tabular data structure. Before Codd published his formative work [3] on this topic in 1970, other approaches were already known, e.g., the hierarchical and the network model [16], all of which had certain shortcomings listed in the paper by Codd. For research purposes, IBM implemented a DBMS called "System R"

Seat			
Number	Train	Date	Reserved
3	RJ65	25-11-2022	Max
...	...	...	...

Table 2.1.: A typical table

based on Codd's relational model [15]. The supplied query language SEQUEL can be seen as the predecessor of SQL. Commercial systems, such as SQL/DS (1981), were developed by IBM. More details can be found in [17]. In fact, several sublanguages exist [15], such as

- DQL (Data Query Language),
- DDL (Data Definition Language) and
- DML (Data Manipulation Language).

This thesis exclusively deals with the DQL, i.e., with those aspects of the language that deal with data retrieval without changing or enhancing them.

Terminology

A relational database consists of a collection of tables. It is possible to refer to the columns of a table (also called *attributes*) via names [15]. Sometimes it makes sense to use a column's fully qualified name, i.e., its name, along with that of the parent table.

It is assumed that all cell entries in a table are atomic in the sense that they do not themselves contain data in tabular form. Codd also demands this and calls it the *first normal form* of a relation [3]. The term *relation* is a mathematical term, but it can be identified with that of a table. A relation is a subset of the Cartesian product of so-called *domains*, i.e., sets from which the entries in a specific column may stem. A single table row is regarded as an element of the associated relation and is called a *tuple*.

2.2. Relational Algebra

In addition to other approaches, such as the relational tuple calculus or the domain calculus [15], relational algebra (RA) is a formal, mathematical foundation for the relational model. It can be shown that the relational tuple calculus and the domain calculus are as powerful as RA, i.e., these systems are relationally complete (Codd's Theorem [4]). Therefore, query languages use concepts from one of the three approaches as needed. In the case of SQL, this is RA, but concrete implementations often differ in some respects from the initial mathematical definitions. One manifestation of this is the inclusion of syntactic sugar, i.e., language components that are added, although the related concepts can be already expressed in the existing language, albeit less comfortably. What is much more remarkable here is that, in some implementations of RA, for example, in SQLite [18], rows can appear multiple times in a table. However, this is precluded by a fundamental assumption of RA that all relations are sets. Nonetheless, it is not difficult to find the main components [19] of RA in its implementations, as illustrated below:

- *Set operations:* Union, intersection, complement, cartesian product
- *Selection:* Filtering on rows
- *Projection:* Filtering on columns
- *Renaming:* Temporary change of a table or column name
- *Join:* Linking tables based on attributes

For the subsequent demonstrations of the concepts, we use Chinook [20], a database of a fictitious media store.

Set operations

The basic set operations are implemented using subqueries and the keywords UNION and INTERSECT. In relational algebra, it is required that the arity of the relations and also the domains of the respective columns match [15]. In SQLite, only arity matters. An example of a union is demonstrated in Listing 2.1, which works analogously for the intersection.

```
sqlite> SELECT FirstName, LastName FROM employees
   ...> UNION
   ...> SELECT FirstName, LastName FROM customers;
Aaron|Mitchell
Alexandre|Rocha
Andrew|Adams
Astrid|Gruber
Bjørn|Hansen
Camille|Bernard
Daan|Peeters
...
```

Listing 2.1: SQL union example

In particular, the result of this query is a list of all full names of both employees and customers.

Selection

If we want to extract all employees belonging to the IT staff of the store, we can use the query given in Listing 2.2. The full output is in the form of a table. This query corresponds exactly to relational algebraic selection.

```
sqlite> SELECT * FROM employees WHERE Title = "IT Staff";
7|King|Robert|IT Staff|6|1970-05-29 00:00:00|2004-01-02 00:00:00|590
    Columbia Boulevard West|Lethbridge|AB|Canada|T1K 5N8|+1 (403)
    456-9986|+1 (403) 456-8485|robert@chinookcorp.com
8|Callahan|Laura|IT Staff|6|1968-01-09 00:00:00|2004-03-04 00:00:00|923 7
    ST NW|Lethbridge|AB|Canada|T1H 1Y8|+1 (403) 467-3351|+1 (403) 467-8772|
    laura@chinookcorp.com
```

Listing 2.2: SQL selection example

The result is a list of all employees of the IT staff. The asterisk (*) denotes that we want all the information about these employees.

Projection

Somewhat confusingly, the projection in SQL is done with the SELECT keyword (see Listing 2.3). In the previous example, a projection was already carried out implicitly, whereby all attributes of the table are used due to the asterisk ($*$). At this point, it is now possible to restrict oneself to specific columns.

```
sqlite> SELECT LastName, FirstName, Address FROM employees;
Adams|Andrew|11120 Jasper Ave NW
Edwards|Nancy|825 8 Ave SW
Peacock|Jane|1111 6 Ave SW
Park|Margaret|683 10 Street SW
Johnson|Steve|7727B 41 Ave
Mitchell|Michael|5827 Bowness Road NW
King|Robert|590 Columbia Boulevard West
Callahan|Laura|923 7 ST NW
```

Listing 2.3: SQL projection example

The result is a list of all employees, but we restrict the output to contain only their full names and their addresses.

Renaming

Renaming tables and their columns within a query may seem superfluous at first. However, it becomes crucial when a table is to be joined to itself (see subsequent paragraph *Join*). Then the uniqueness of a designation is in danger. How this works can be seen in Listing 2.5.

```
sqlite> SELECT EMP.FirstName AS fn FROM employees AS EMP;
Andrew
Nancy
Jane
Margaret
Steve
Michael
Robert
Laura
```

Listing 2.4: SQL renaming example

Technically, the query result in Listing 2.4 would be the same without the `AS` clauses, but it demonstrates how renaming is done.

Join

The most common type of joins in SQL is the Equi-Join, which is based on the θ join of relational algebra, where θ is the applied equality operator, which is usually $=$, but it can also be, for instance, $\neq$, $<$ or $\leq$ [21]. All rows from two tables are merged into one by comparing the values of two attributes; if the comparison succeeds, the concatenation of the two rows is added to the resulting relation. An example of a join is shown in Listing 2.5:

```
sqlite> SELECT emp1.FirstName, emp1.LastName, emp2.FirstName, emp2.LastName
   ...> FROM employees AS emp1 INNER JOIN employees AS emp2
   ...> ON emp1.EmployeeId = emp2.ReportsTo;
Andrew|Adams|Nancy|Edwards
Nancy|Edwards|Jane|Peacock
Nancy|Edwards|Margaret|Park
Nancy|Edwards|Steve|Johnson
```

```
Andrew|Adams|Michael|Mitchell
Michael|Mitchell|Robert|King
Michael|Mitchell|Laura|Callahan
```

Listing 2.5: SQL join example

The result of this join is a table whose rows contain the full name of two employees each, such that the second employee reports to the first employee.

2.3. Program Verification and Model Checking (WIP)

This section discusses which formal methods are available for verifying theorems and programs, and how they differ from each other.

2.3.1. Formal Verification

Our daily lives depend on software that works as expected. This is particularly true in safety-critical areas such as healthcare, cryptography, and aerospace. To achieve this, it is necessary to prove mathematically that an algorithm is correct, i.e., that it conforms to a formal specification, which can be thought of as the proof of a theorem.

Proof-based Verification

Throughout history, there have been several attempts to automate mathematical proofs, such as the *Principia Mathematica* (published between 1910 and 1930) by Whitehead and Russell [22]. Many other formal systems are equivalent to PM. Automated provers can prove many theorems of interest [23].

One of the advantages of modern automated theorem provers is the expressiveness of their language and their generality. For example, they typically run on first-order logic (FOL) variants and support quantification over infinite domains [19].

FOL is an example for a so-called *deductive system* [23], [24], that consists of

- an *alphabet* Ω of symbols like constants, predicate symbols, and function symbols,

- a set $\mathfrak{A}$ of *axioms*, i.e., formulas that consist of symbols in Ω, and
- a set of *inference rules*, that describe how a new formula ψ can be derived from from an existing one ϕ. If there is such a rule, we write $\phi \vdash \psi$.

The *language* of a deductive system contains all its axioms and is closed under the application of inference rules. A *proof* of a formula ϕ is a chain of formulas $(\phi_i)_{i=0}^{n}$, s.t. $\phi_0 \in \mathfrak{A}$, $\phi_n = \phi$ and

$$\phi_0 \vdash \phi_1 \vdash \cdots \vdash \phi_n.$$

If a proof for ϕ exists, we write for short $\vdash \phi$.

A proving deductive system lies on the level of syntax, as it is just a manipulation of symbols according to rules. The *semantics* describe the truth value of every formula of the language through an interpretation I of each symbol [7]. If a formula ϕ is true with respect to I, we write $I \models \phi$. If this is the case independently of the interpretation, we call this formula *valid* and write instead $\models \phi$.

On the one hand, we want the language of a deductive system to contain only true formulas (i.e., $\vdash \phi \Rightarrow \models \phi$). If this is given, we call it *sound*. On the other hand, every true formula should be in the language (i.e., $\models \phi \Rightarrow \vdash \phi$). In this case, it is called *complete*.

FOL has a deductive system that is both sound and complete (i.e., $\models \phi \Leftrightarrow \vdash \phi$) [23], which is good, but Turing proved that FOL is undecidable [24]. This means that an effective procedure does not exist that tells us whether a formula is valid. However, it is still *semi-decidable*, i.e., there is a procedure that confirms the validity of a formula, but for invalid formulas, it loops forever [23]. Consequently, we can hardly tell if a formula is either true and the procedure takes very long or if it is just false.

A *first order theory* [7], [24] is based on the deductive system of FOL and gets enhanced by additional symbols (e.g., constants, functions, predicates) and additional axioms (*proper axioms*). From a *consistent* theory, no contradictory formulas can be derived (i.e., either $\vdash \phi$ or $\vdash \neg\phi$). Gödel proved in his incompleteness theorems that no theory that axiomatizes arithmetic (e.g., Peano axioms) can be complete and consistent simultaneously.

There is a method, namely the *Hoare Calculus* [7], to create *verification conditions* for a program. These are logical formulas that can be proven automatically and imply the correctness of the program. They are created based on certain *annotations* a human writes for the program, e.g., *pre-conditions* (should be given before execution), *Post-conditions* (should be given after execution) and loop invariants (conditions that hold in every iteration of a loop).

Invalid verification conditions are produced if there are errors in

- the specification of the algorithm
- or the annotations.

As told earlier, this leads to Non-Termination (compare semi-decidability). However, it also can fail because the verification condition is not provable within the calculus.

Failure to generate a proof does not necessarily imply that the goal of the proof is wrong but that the method used needs to be revised [19].

In program analysis and verification, it is unnecessary to model variables by unbounded data types since data types in programming languages such as C are bounded [22]. This is where model checking techniques can be used.

Model Checking

The goal of *decidability* can be achieved by restricting integer values to a specific size [25]. A model checker provides an effective decision procedure to verify that the program conforms to the specification. This is a particular case of the verification of a *finite transition system* (Kripke structure) where the specification is expressed in *Linear Temporal Logic* (LTL) [22].

If the model checker reports an error, it presents a counterexample as a system trace that violates the specification. The first model checker was introduced by Clarke and Emerson and attempted to traverse the state space of the system (*Explicit-state Model Checker*). Although there are only finitely many states, there may be too many to process in a reasonable time (*state-explosion problem*). The main goal is, therefore, to reduce the search space. Another approach to the problem is *Symbolic Model Checking*, where the set of states is represented by a *Binary Decision Diagram* (BDD), which improves performance but may also consume a large amount of memory [26].

The idea of *Bounded Model Checking* (BMC) is not to verify a theorem but to falsify it, i.e., to find a counterexample, while restricting itself to program runs of finite length [26]. Such a problem can be transformed so that it can be solved by an SAT solver or an SMT solver [22], which we will consider in the next section.

SAT/SMT solving

The Boolean Satisfiability Problem (SAT) [23] can be stated as follows: Given a propositional formula ψ, is there a replacement of the propositional variables by truth values such that ψ becomes true? Or, in other words: Is ψ satisfiable? A large variety of problems can be reduced to SAT; hence, many efficient solvers have been developed [12].

We know that first order logic (FOL) is generally undecidable, unlike propositional logic [23]. FOL contains quantifiers, predicate symbols, and function symbols, which can have multiple (non-standard) interpretations. In many cases, this is not necessarily such that we may fix a specific meaning of the symbols, e.g., $<$ is the linear order of the integers. This is the idea of SMT (Satisfiability Modulo Theories) [23]. There is a group called *SMT-LIB* initiative [27], which is concerned with creating standards for the SMT community, e.g.

- a same-named input/output language with a LISP-like syntax and
- multiple theories like
  - QF_LIA, QF_LRA (quantifier-free linear integer/real arithmetic),
  - QF_UFBV (quantifier-free formulas over bit vectors with uninterpreted sort function and symbols) and many more.

Depending on the theory, SMT solvers can use SAT solvers in the backend. Model checkers can benefit from the capabilities of SMT because finite domains can be encoded as bit vectors, in the sense of the theory QF_UFBV: For $n \in \mathbb{N}$, any natural number $m \leq n$ can be represented by a bit vector of length $\left\lceil \frac{log(n)}{log(2)} \right\rceil$; every set S of natural numbers less than or equal to n, can be represented by a bit vector of length $n + 1$, where the i-th bit is 1 if $i \in S$.

2.3.2. RISCAL

The RISC Algorithm Language (RISCAL) [9] comprises a specification language and a software system, one of the primary purposes of which is to assist students with proof and program verification problems [28].

Unlike low-level programming languages such as C, the RISCAL language is based on a typed variant of first-order logic [29] and is therefore designed to model algorithms at a high level of abstraction [10], [28]. Thus it supports the following:

- a large variety of *data types* and
- specialized *operations* such as quantified formulas as boolean literals and the ability to choose a value that satisfies a formula non-deterministically.

In RISCAL, the *state space* of systems or programs and the universe of quantified formulas are finite [28]. This is because the values of variables are restricted to finite types that can depend on a parameter set manually by the user so that each formal specification in RISCAL represents an infinite class of finite models [10], [29].

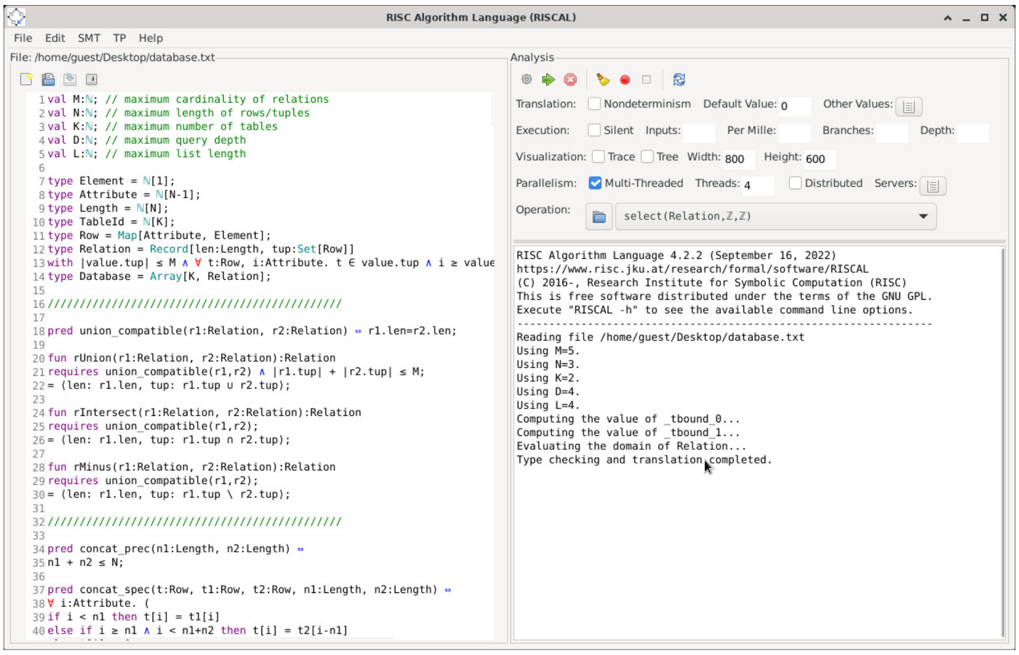
RISCAL allows annotating an algorithm with formulas such as:

- *Pre- and postconditions:* What input arguments are allowed, and what is considered a correct result?
- *Loop invariants:* What conditions must hold in every loop iteration?

These can be checked by running the program (*runtime assertion checking*) for all feasible inputs. If it succeeds, the assertion is *validated*; otherwise, it is *falsified* (note that *validation* is different from *verification* since it means proving the conjecture for all (infinitely many) finite domains [10]).

However, RISCAL also generates verification conditions, i.e., formulas whose validity implies the correctness of these annotations. RISCAL implements two alternatives to check the validity of a formula:

- *Semantic evaluation:* With an executable variant of the denotational semantics of every RISCAL expression, a formula can be evaluated to determine its truth value.


RISC Algorithm Language (RISCAL)
File
Edit
SMT
TP
Help
File: /home/guest/Desktop/database.txt
Analysis

```
val M:ℕ; // maximum cardinality of relations
val N:ℕ; // maximum length of rows/tuples
val K:ℕ; // maximum number of tables
val D:ℕ; // maximum query depth
val L:ℕ; // maximum list length

type Element = ℕ[1];
type Attribute = ℕ[N-1];
type Length = ℕ[N];
type TableId = ℕ[K];
type Row = Map[Attribute, Element];
type Relation = Record[len:Length, tup:Set[Row]]
with |value.tup| ≤ M ∧ ∀ t:Row, i:Attribute. t ∈ value.tup ∧ i ≥ value
type Database = Array[K, Relation];

/////////////////////////////////////////////

pred union_compatible(r1:Relation, r2:Relation) ⇔ r1.len=r2.len;

fun rUnion(r1:Relation, r2:Relation):Relation
requires union_compatible(r1,r2) ∧ |r1.tup| + |r2.tup| ≤ M;
= (len: r1.len, tup: r1.tup ∪ r2.tup);

fun rIntersect(r1:Relation, r2:Relation):Relation
requires union_compatible(r1,r2);
= (len: r1.len, tup: r1.tup ∩ r2.tup);

fun rMinus(r1:Relation, r2:Relation):Relation
requires union_compatible(r1,r2);
= (len: r1.len, tup: r1.tup \ r2.tup);

/////////////////////////////////////////////

pred concat_prec(n1:Length, n2:Length) ⇔
n1 + n2 ≤ N;

pred concat_spec(t:Row, t1:Row, t2:Row, n1:Length, n2:Length) ⇔
∀ i:Attribute. (
if i < n1 then t[i] = t1[i]
else if i ≥ n1 ∧ i < n1+n2 then t[i] = t2[i-n1]
```

Translation: ☐ Nondeterminism Default Value: 0 Other Values:
Execution: ☐ Silent Inputs: Per Mille: Branches: Depth:
Visualization: ☐ Trace ☐ Tree Width: 800 Height: 600
Parallelism: ☑ Multi-Threaded Threads: 4 ☐ Distributed Servers:
Operation: select(Relation,ℤ,ℤ)

```
RISC Algorithm Language 4.2.2 (September 16, 2022)
https://www.risc.jku.at/research/formal/software/RISCAL
(C) 2016-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCAL -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /home/guest/Desktop/database.txt
Using M=5.
Using N=3.
Using K=2.
Using D=4.
Using L=4.
Computing the value of _tbound_0...
Computing the value of _tbound_1...
Evaluating the domain of Relation...
Type checking and translation completed.
```

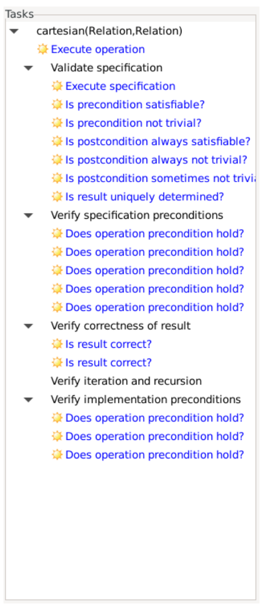
Figure 2.1.: The RISCAL User Interface

- *SMT-LIB translation:* By translating the formula into the decidable theory QF_UFBV, many different SMT solvers (Boolector, Yices, Z3, CVC4) can be used for checking the validity.

Figure 2.1 shows the user interface of RISCAL. On the left side, one can see the specification editor; on the right side are options, controls, and the output window.

The menu with the various conditions generated by RISCAL for verifying an algorithm is shown in Figure 2.2.

Since 2022 RISCAL also supports (via the RISCTP interface to external theorem provers) the proof of formulas over domains of arbitrary size. The RISC Theorem Proving Interface (RISCTP) [13] consists of a language, modeled on both RISCAL and SMT-LIB, for specifying proof problems and an interface to various provers (Z3, CVC5, Vampire). Its integration into the RISCAL environment enriches its model checking capabilities because such a proof verifies the validity of a theorem for all finite models [14].



```
Tasks
▾ cartesian(Relation,Relation)
    Execute operation
  ▾ Validate specification
      Execute specification
      Is precondition satisfiable?
      Is precondition not trivial?
      Is postcondition always satisfiable?
      Is postcondition always not trivial?
      Is postcondition sometimes not trivi
      Is result uniquely determined?
  ▾ Verify specification preconditions
      Does operation precondition hold?
      Does operation precondition hold?
      Does operation precondition hold?
      Does operation precondition hold?
      Does operation precondition hold?
  ▾ Verify correctness of result
      Is result correct?
      Is result correct?
    Verify iteration and recursion
  ▾ Verify implementation preconditions
      Does operation precondition hold?
      Does operation precondition hold?
      Does operation precondition hold?
```

Figure 2.2.: Tasks for a single operation with pre- and postcondtions

2.4. Formal Languages

The query language presented in Chapter 4 is a formal language. It is, therefore, worth familiarizing ourselves with the concepts of formal languages, namely the two key components:

- *Syntax:* How does a correct sentence of the language look like?
- *Semantics:* What is the interpretation of a sentence?

2.4.1. Abstract Syntax

In this section, we define what *abstract syntax* is and look at a small example. The word "abstract" refers to the fact that in this approach, rather, the structure of a syntactic phrase is defined, not its representation in particular (e.g., as a string).

Definition 2.1 *An **abstract syntax** [7] has two components. The first one is the declaration of one or more **syntactial domains***

$$D \quad \in \quad \mathit{Domain}$$

where the left-hand and right-hand side is uniquely named. D is a ***variable*** *or more accurately a* ***non-terminal symbol*** *and denotes an element of* Domain*. For each declaration, there are one or more* ***production rules***

$$D \quad ::= \quad \mathrm{T}_1(N_{1,1}, \ldots, N_{1,n_1}) \;\mid\; \cdots \;\mid\; \mathrm{T}_m(N_{m,1}, \ldots, N_{1,n_m})$$

where $n_i \leq 0$. The symbols T_i are called ***terminal symbols****, and the variables in the parentheses are arbitrary* ***non-terminal symbols****. Together they form an* ***alternative****.*

The notation we use is known as the *Backus-Naur form* [7]. A grammar, in the following way, can generate every syntactic expression:

1. Start with the non-terminal symbol in the first declaration.
2. Replace a non-terminal symbol using the production rules.
3. If the phrase still contains non-terminal symbols, repeat step 2.

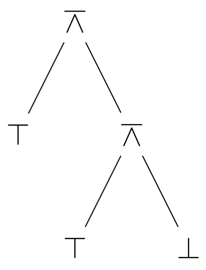
We present a simple example to motivate the previous definition better:

$$\begin{array}{lcl} B & \in & \mathit{Bool} \\ B & ::= & \bot() \mid \top() \mid \overline{\wedge}(B_1, B_2) \end{array}$$

The intuition of this construction is that all phrases represent expressions with $\overline{\wedge}$ (NAND) as binary operator corresponding to an evaluation of a boolean-valued function. The symbols $\top$ and $\bot$ denote true (1) and false (0). Below we let the production rules work on the initial expression B. Note that we can omit parentheses for alternatives consisting of only a terminal like $\bot()$.

$$B \to \overline{\wedge}(B_1, B_2) \to \overline{\wedge}(\overline{\wedge}(B_3, B_4), B_2) \to \overline{\wedge}(\overline{\wedge}(\top, B_4), B_2) \to \cdots \to \overline{\wedge}(\overline{\wedge}(\top, \bot), \top)$$

It should be noted that this formal language only suggests a meaning but still needs to have it. The expression $f := \overline{\wedge}(\top, \overline{\wedge}(\top, \bot))$ can be represented as a so-called AST (Abstract Syntax Tree) in the following way:



Note that this tree is independent of the *concrete syntax* of the expression f, i.e., we can use the infix notation $(B_1 \overline{\wedge} B_2)$ instead of $\overline{\wedge}(B_1, B_2)$. An alternative way to write f is $\top \overline{\wedge} (\top \overline{\wedge} \bot)$. The outermost brackets do not influence the structure of the AST.

2.4.2. Denotational Semantics

One way of giving meaning to a language is called *denotational semantics* [7]. The basic idea is to create a mapping $[\![\cdot]\!]$, that maps every AST of a syntactic domain to a mathematical object (*denotation*) of a semantic domain $\mathcal{D}$.

The technique that makes this possible is *structural induction*. Although it is used to prove properties for all elements of a syntactic domain, it can also be used to define functions on a syntactic domain. The image of such a function can be any set, e.g., the same or another syntactic domain. This is achieved by the definition of the result $[\![A]\!]$ for any alternative A.

Take the domain `Bool` from earlier. We choose as the semantic domain $\mathcal{D} := \{0, 1\} \subset \mathbb{N}$, thus $[\![\cdot]\!] : \texttt{Bool} \to \{0, 1\}$. It is inductively defined by

$$[\![\bot]\!] := 0, \quad [\![\top]\!] := 1, \quad [\![B_1 \overline{\wedge} B_2]\!] := \mathbb{1}_{\{0\}}([\![B_1]\!] \cdot [\![B_2]\!]).$$

For $f := \top \overline{\wedge} (\top \overline{\wedge} \bot)$, we get

$$[\![f]\!] = \mathbb{1}_{\{0\}}([\![\top]\!] \cdot [\![\top \overline{\wedge} \bot]\!]) = \mathbb{1}_{\{0\}}(\mathbb{1}_{\{0\}}([\![\top]\!] \cdot [\![\bot]\!])) = \mathbb{1}_{\{0\}}(\mathbb{1}_{\{0\}}(0)) = \mathbb{1}_{\{0\}}(1) = 0.$$

Now it is evident that this formal system reproduces the NAND operation.

3. Relational Algebra

This chapter presents the formal framework that underlies the rest of the work. We introduce the domain $\texttt{Relation}_{\mathcal{A}}$ and the operations that act on it. Moreover, we discuss selected theorems that hold in relational algebra.

3.1. Mathematical Model

3.1.1. Domain

First Considerations

In the mathematical model, we will consider, for simplicity, infinite domains. Since we are talking about tables, it is essential to know what the cells of the tables contain. We will use a simple collection of domains $U := \{\Omega^*, \mathbb{N}\}$ which allows us to represent strings $\omega \in \Omega^*$ of characters of an alphabet Ω (e.g., ASCII values) and numeric values $a \in \mathbb{N}$.

In practice, it is common to name each column j by an attribute A_j and provide a domain $\mathrm{dom}(A_j) \in U$ for it [15]. A certain set $A = \{A_1, \ldots, A_n\}$ of attributes is called an *attribute schema* or relation schemes [15], [21]. Rows are mostly described as tuples $t \in \mathrm{dom}(A_1) \times \cdots \times \mathrm{dom}(A_n)$ but they can also be treated as functions $t : \{A_1, \ldots, A_n\} \to \bigcup U$ with the restriction $t(A_i) \in \mathrm{dom}(A_i)$ [21]. In database theory, relations are (finite) subsets of $\prod_{i=1}^{n} \mathrm{dom}(A_i)$ [15].

Data Types

Definition 3.1 *Let $n \in \mathbb{N}$ and $\mathcal{A} = \{A_1, \ldots, A_n\}$ be an attribute schema. Then the set*

$$\texttt{Row}_{\mathcal{A}} := \left\{ t : A \to \bigcup_{i=1}^{n} \operatorname{dom}(A_i) : t(A_i) \in \operatorname{dom}(A_i) \textit{ for all } 1 \le i \le n \right\}$$

contains all tuples. Furthermore the set

$$\texttt{Relation}_{\mathcal{A}} := \{ r \subseteq \texttt{Row}_{\mathcal{A}} : |r| < \infty \}$$

contains all relations.

The index $\mathcal{A}$ in both definitions means that they conform to schema $\mathcal{A}$. In the case that $n = 0$, we introduce the convention that $\mathcal{A} = \emptyset$ and therefore the only tuple in $\texttt{Row}_{\mathcal{A}}$ is $t : \emptyset \to \emptyset$, which we itself denote as $\emptyset$. Hence, $\texttt{Relation}_{\mathcal{A}} = \{\emptyset, \{\emptyset\}\}$.

3.1.2. Operations

The operations of our model are functions of which each operates on one or more relations and create a new relation as its result. The simplest ones are the binary set operations because these are to be understood in the generally known way. We will therefore omit their definitions but refer the reader to [1].

In this section following, we will define the set operations *Cartesian Product* and *join* based on the following auxiliary function acting on rows.

Definition 3.2 *Let $\mathcal{A} = \{A_1, \ldots, A_n\}$, $\mathcal{B} = \{B_1, \ldots, B_m\}$ be relation schemas. The function $c : \texttt{Row}_{\mathcal{A}} \times \texttt{Row}_{\mathcal{B}} \to \texttt{Row}_{\mathcal{A} \cup \mathcal{B}}$ given by*

$$c(t, u) := (t(A_1), \ldots, t(A_n), u(B_1), \ldots, u(B_m))$$

is called the concatenation of the rows t and u.

Definition 3.3 *Let $\mathcal{A} = \{A_1, \dots, A_n\}$, $\mathcal{B} = \{B_1, \dots, B_m\}$ be attribute schemas. The Cartesian product C : $\texttt{Relation}_\mathcal{A} \times \texttt{Relation}_\mathcal{B} \to \texttt{Relation}_{\mathcal{A}\cup\mathcal{B}}$ is given by*

$$C(r, s) := \{c(t, u) : t \in r \wedge u \in s\}.$$

Definition 3.4 *Let $\mathcal{A} = \{A_1, \dots, A_n\}$, $\mathcal{B} = \{B_1, \dots, B_m\}$ be attribute schemas. The function $\bowtie$: $\texttt{Relation}_\mathcal{A} \times \texttt{Relation}_\mathcal{B} \times \mathcal{A} \times \mathcal{B} \to \texttt{Relation}_{\mathcal{A}\cup\mathcal{B}}$ given by*

$$\bowtie (r, s, A, B) := \{c(t, u) : t \in r \wedge u \in s \wedge t(A) = u(B)\}$$

is called the (equi-)join of r and s on A and B.

Join and the Cartesian product are connected through the following theorem [21].

Theorem 3.1 *Let $\mathcal{A}, \mathcal{B}$ be attribute schemas, $A \in \mathcal{A}$, $B \in \mathcal{B}$ and $r \in \texttt{Relation}_\mathcal{A}$, $s \in \texttt{Relation}_\mathcal{A}$ relations. Then*

$$\bowtie (r, s, A, B) \subseteq C(r, s)$$

holds.

The following operation filters out rows from a relation, namely those with a specific value at one attribute.

Definition 3.5 *Let $\mathcal{A} = \{A_1, \dots, A_n\}$ be an attribute schema. Then the function σ : $\texttt{Relation}_\mathcal{A} \times \mathcal{A} \times \bigcup_{A\in\mathcal{A}} \mathrm{dom}(A) \to \texttt{Relation}_\mathcal{A}$ given by*

$$\sigma(r, A, a) := \{t \in r : t(A) = a\}$$

is called the selection of all rows t in r with value a at attribute A.

Several properties of the selection can be observed. We will look at two examples, namely the compatibility of the selection with set operations and the commutativity of the composed selections [21].

Theorem 3.2 *Let $\mathcal{A}$ be an attribute schema, $A \in \mathcal{A}$, $a \in \operatorname{dom}(A)$ and $r, s \in \texttt{Relation}_{\mathcal{A}}$ be relations. Then*

$$\sigma(r \,\gamma\, s, A, a) = \sigma(r, A, a) \,\gamma\, \sigma(s, A, a)$$

holds for every $\gamma \in \{\cup, \cap, \setminus\}$.

Because the intersection is commutative, the last theorem implies that the composition of selections is commutative [21].

Theorem 3.3 *Let $\mathcal{A}$ be an attribute schema, $A, B \in \mathcal{A}$, $a \in \operatorname{dom}(A), b \in \operatorname{dom}(B)$ and $r \in \texttt{Relation}_{\mathcal{A}}$ be a relation. Then*

$$\sigma(\sigma(r, B, b), A, a) = \sigma(r, A, a) \cap \sigma(r, B, b)$$

holds.

The projection takes a relation as input and creates a new one on a subschema containing the same information.

Definition 3.6 *Let $\mathcal{A} = \{A_1, \ldots, A_n\}$ be an attribute schema. The operation $\pi : \texttt{Relation}_{\mathcal{A}} \times \mathcal{P}(\mathcal{A}) \to \bigcup_{\mathcal{A}' \subseteq \mathcal{A}} \texttt{Relation}_{\mathcal{A}'}$ defined by*

$$\pi(r, \mathcal{B}) = \{t|_{\mathcal{B}} : t \in r\}$$

is called projection of r on $\mathcal{B}$.

3.2. RISCAL Model

The requirements for a theory to be processed with the computer are that all objects must be assigned to a predefined data type, and functions should have a formal description. We will use two methods for this. Implicit descriptions consist of conditions for the inputs and the result of a function, whereas an explicit or algorithmic description is a procedure for computing the result.

3.2.1. Parameters

The sets $\texttt{Row}_{\mathcal{A}}$ and $\texttt{Relation}_{\mathcal{A}}$ can have arbitrary dimensions, i.e., the number of rows and columns can be any positive integer. In order to make the domains amenable to a model checker, we need to introduce some bounds. Since a database is a collection of tables, we also have a bound for the number of tables. Furthermore, a query can be a part of another query. How many queries can be nested into each other also needs to be bounded. All these variables are shown in Listing 3.1.

```
val M:ℕ; // maximum cardinality of relations
val N:ℕ; // maximum length of rows/tuples
val K:ℕ; // maximum number of tables
val D:ℕ; // maximum query depth
```

Listing 3.1: Bounds for the data types

3.2.2. Types

In an earlier section, we assumed that domains of attributes are either of Ω^* or $\mathbb{N}$. If we continued with this approach, it would lead to large amounts of time necessary for model checking. Hence, we simplify this and assume every domain as $\{0, 1\}$ (line 1 in the Listing 3.2). Also, a relation can have at most N attributes, and the type `Attribute` contains attribute indices from 0 to $N - 1$ (line 2). We use a similar type `Length` for the actual number of attributes, reaching from 0 to N (line 3).

```
1 type Element = ℕ[1];
2 type Attribute = ℕ[N-1];
3 type Length = ℕ[N];
```

Listing 3.2: Element and index types

In analogy to $\texttt{Row}_{\mathcal{A}}$, the RISCAL type `Row` is a map with keys in `Attribute` and values in `Element` (see line 1 of Listing 3.3). The RISCAL type `Relation` is also based on $\texttt{Row}_{\mathcal{A}}$, but in addition to a set of rows `tup`, we store the arity `len` of the relation (line 2).

Elements of the type `Relation` must meet certain conditions. Firstly, because M is the maximum number of rows, the cardinality of the `tup` component must have M as its upper bound. For performance reasons, we further introduce the restriction that components of tuples, which at an attribute greater or equal `len`, are zero.

```
type Row = Map[Attribute, Element];
type Relation = Record[len:Length, tup:Set[Row]]
with |value.tup| ≤ M ∧ ∀ t:Row, i:Attribute. t ∈ value.tup ∧ i ≥ value.len
    ⇒ t[i] = 0;
```

Listing 3.3: RISCAL implementation of $\texttt{Row}_{\mathcal{A}}$ and $\texttt{Relation}_{\mathcal{A}}$

In RISCAL, no dynamically growing collections are built in, but one can be constructed with *recursive types* [8]. The meaning of the Listing 3.4 below is as follows: A list is either empty, i.e., is the constant `List!nil` or is a composition of the functions `List!node`.

```
    rectype(D) List = nil | node(Attribute, List);
```

Listing 3.4: List data type

3.2.3. Set operations

For more readability, we have a predicate, shown in Listing 3.5, which denotes pairs of relations suitable for the set operations subsequently introduced.

```
pred union_compatible(r1:Relation, r2:Relation) ⇔ r1.len=r2.len;
```

Listing 3.5: Predicate for union compatibility

RISCAL already provides set operations, which we use to define those for relations (see Listing 3.6). Note that the two input relations need to be union compatible, and in the case of union, the maximum number of rows could be exceeded. We take care of this by adding the precondition that the sum of cardinalities of the relations shall have the upper bound M.

```
fun rUnion(r1:Relation, r2:Relation):Relation
requires union_compatible(r1,r2) ∧ |r1.tup| + |r2.tup| ≤ M;
= ⟨len: r1.len, tup: r1.tup ∪ r2.tup⟩;

fun rIntersect(r1:Relation, r2:Relation):Relation
```

```
requires union_compatible(r1,r2);
= ⟨len: r1.len, tup: r1.tup ∩ r2.tup⟩;

fun rMinus(r1:Relation, r2:Relation):Relation
requires union_compatible(r1,r2);
= ⟨len: r1.len, tup: r1.tup \ r2.tup⟩;
```

Listing 3.6: Implementation of the set operations

3.2.4. Cartesian Product

In Listing 3.7, the precondition for the `concat` function is that the length of the two tuples in the input does not exceed the maximum arity. Because c is an operation on $\mathtt{Row}_{\mathcal{A}} \times \mathtt{Row}_{\mathcal{A}}$, our implementation also takes two rows, but in addition to that, the arities of these two tuples, i.e., for indices not exceeding `n1` the components of the output are taken from `t1`. For indices not exceeding `n1+n2`, the tuple progresses with the components of `t2`. The right side of the output is padded with zeros.

```
pred concat_prec(n1:Length, n2:Length) ⇔
n1 + n2 ≤ N;

pred concat_spec(t:Row, t1:Row, t2:Row, n1:Length, n2:Length) ⇔
∀ i:Attribute. (
if i < n1 then t[i] = t1[i]
else if i ≥ n1 ∧ i < n1+n2 then t[i] = t2[i-n1]
else t[i] = 0
);
```

Listing 3.7: Pre- and postconditions for c

The most straightforward "implementation" of this "specification" consists of a `choose` expression, that tells RISCAL to return a result that fulfills the postcondition `concat_spec` (see Listing 3.8).

```
fun concat1(t1:Row, t2:Row, n1:Length, n2:Length):Row
requires concat_prec(n1, n2);
= choose t:Row with concat_spec(t,t1,t2,n1,n2);
```

Listing 3.8: Implicit implementation of c

The procedure for the concatenation has the same signature and precondition, but as its postcondition, it ensures that it is equivalent to the specification `concat_spec`. In line 4 of Listing 3.9, a bit array of size N, with entries only containing zero, is created. The same as before is now achieved by two loops at the lines 5 and 8.

```
 1 proc concat2(t1:Row, t2:Row, n1:Length, n2:Length):Row
 2 requires concat_prec(n1, n2);
 3 ensures concat_spec(result,t1,t2,n1,n2); {
 4     var t:Row := Array[N,Element](0);
 5     for var i:Length:=0; i<n1; i:=i+1 do {
 6         t[i] := t1[i];
 7     }
 8     for var i:Length:=n1; i<n1+n2; i:=i+1 do {
 9         t[i] := t2[i-n1];
10     }
11     return t;
12 }
```

Listing 3.9: Explicit implementation of c

The following function definition (see Listing 3.10) is an alias for the previous one. Also, those parts of the source code will be omitted. Every procedural implementation has such an alias without a digit at the end.

```
fun concat(t1:Row, t2:Row, n1:Length, n2:Length):Row
requires concat_prec(n1, n2);
ensures concat_spec(result,t1,t2,n1,n2);
= concat2(t1, t2, n1, n2);
```

Listing 3.10: Alias as main implementation

The `choose` keyword provides a way to express the Cartesian product using quantifiers. In Listing 3.11, it takes a logical formula as input, specifying the resulting rows' desired properties. Essentially, the `choose` keyword allows us to select rows from the input tables such that their concatenation satisfies the given formula.

```
pred cartesian_prec(r1:Relation, r2:Relation) ⇔
r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;

pred cartesian_spec(r:Relation, r1:Relation, r2:Relation) ⇔
```

```
r.len = r1.len+r2.len ∧
∀ t:Row. t∈r.tup ⇔ ∃ t1:Row, t2:Row.
    t1∈r1.tup ∧ t2∈r2.tup ∧ concat_spec(t,t1,t2,r1.len,r2.len);
```

Listing 3.11: Pre- and postconditions for *C*

The first thing one may think of when defining the Cartesian product is to use the built-in operator $\times$ for the Cartesian product of sets. The problem is that we need to merge any two rows into one instead of just creating pairs. With RISCAL, we can use the implicit set definition, i.e., $\{A(e) : e \in S\}$ is the set of all $e \in S$ with the property A. This approach is visible in Listing 3.12.

```
fun cartesian2(r1:Relation, r2:Relation):Relation
requires cartesian_prec(r1, r2);
ensures cartesian_spec(result,r1,r2);
= ⟨len: r1.len+r2.len, tup: {concat(t1,t2,r1.len,r2.len) | t1∈r1.tup, t2∈r2
    .tup}⟩;
```

Listing 3.12: Explicit implementation of *C* (version 1)

The algorithm for computing the Cartesian product is straightforward (see Listing 3.13). In two nested loops, the concatenations of any two rows are collected.

```
    proc cartesian3(r1:Relation, r2:Relation):Relation
    requires cartesian_prec(r1, r2);
    ensures cartesian_spec(result,r1,r2); {
        var q:Relation := ⟨len: r1.len+r2.len, tup: choose s:Set[Row] with
            |s|=0⟩;

        for t1 ∈ r1.tup do {
            for t2 ∈ r2.tup do {
                q.tup := q.tup ∪ {concat(t1, t2, r1.len, r2.len)};
            }
        }

        return q;
    }
```

Listing 3.13: Explicit implementation of *C* (version 2)

3.2.5. Selection

The selection of rows of a relation r with value e at attribute a is of the same arity as the outgoing relation. Membership of rows t in the output s can be characterized by the property that t is in r and has the property $t(a) = e$. An obvious precondition is to ensure that the attribute a does not exceed the arity of r (see Listing 3.14).

```
pred select_prec(a:Attribute) ⇔
a < r.len;

pred select_spec(s:Relation, r:Relation, a:Attribute, e:Element) ⇔
s.len = r.len ∧ ∀ t:Row. t∈s.tup ⇔ t∈r.tup ∧ t[a] = e;
```

Listing 3.14: Pre- and postconditions for σ

This can also be written more explicitly with the implicit set definition shown in Listing 3.15.

```
fun select2(r:Relation, a:Attribute, e:Element):Relation
requires select_prec(a);
ensures select_spec(result,r,a,e);
= ⟨len: r.len, tup: {t | t∈r.tup with t[a] = e}⟩;
```

Listing 3.15: Explicit implementation of σ (version 1)

An algorithmic description is also possible. In Listing 3.16, we iterate over all input rows and collect those with the preferred property in the output.

```
proc select3(r:Relation, a:Attribute, e:Element):Relation
requires select_prec(a);
ensures select_spec(result,r,a,e); {
    var q:Relation := ⟨len: r.len, tup: choose s:Set[Row] with |s|=0⟩;

    for t ∈ r.tup do {
        if t[a] = e then {
            q.tup = q.tup ∪ {t};
        }
    }
    return q;
}
```

Listing 3.16: Explicit implementation of σ (version 2)

Theorems

In RISCAL, theorems are a certain kind of predicate, which are assumed to be true for all values of their parameters. All of the assumptions of a mathematical theorem are coded as preconditions with the `requires` clause. Note that this condition is a conjunction of the preconditions of any function used to formulate the proposition of the theorem. The compatibility property of the operation σ and set operations is implemented in Listing 3.17.

```
theorem select_union_equiv(r1:Relation, r2:Relation, a:Attribute, e:Element
    )
requires a < r1.len ∧ a < r2.len ∧ union_compatible(r1,r2) ∧ |r1.tup| + |r2
    .tup| ≤ M; ⇔
select2(rUnion(r1,r2),a,e) = rUnion(select2(r1, a, e), select2(r2, a, e));

theorem select_intersect_equiv(r1:Relation, r2:Relation, a:Attribute, e:
    Element)
requires a < r1.len ∧ a < r2.len ∧ union_compatible(r1,r2); ⇔
select2(rIntersect(r1,r2),a,e) = rIntersect(select2(r1, a, e), select2(r2,
    a, e));

```

```
theorem select_minus_equiv(r1:Relation, r2:Relation, a:Attribute, e:Element)
requires a < r1.len ∧ a < r2.len ∧ union_compatible(r1,r2); ⇔
select2(rMinus(r1,r2),a,e) = rMinus(select2(r1, a, e), select2(r2, a, e));
```

Listing 3.17: Implementation of Theorem 3.1

The theorem below states that for executions of two selects, composition and intersection are essentially the same (see Listing 3.18).

```
theorem select_intersect_comp(r:Relation, a:Attribute, e:Element, b:Attribute, f:Element)
requires a < r.len ∧ b < r.len; ⇔
select2(select2(r, a, e), b, f) = rIntersect(select2(r, a, e), select2(r, b, f));
```

Listing 3.18: Implementation of Theorem 3.2

3.2.6. Projection

The input must have a specific format to declare a projection pattern. One can argue that this array should contain values of type `Attribute`, but this special format requires an "invalid index". In other words, the left part of the array contains indices in the range of the arity of the input relation, and the right side is padded with the invalid index N (see line 2 of Listing 3.19). Therefore the arity of the output is the number of non-N entries of `columns`. For all rows in the input, there must be a row in the output such that the `i`-th entry of this tuple equals the `columns[i]`-th entry of the input.

```
1 pred project_prec(r:Relation, columns:Array[N,Length]) ⇔
2 ∃ i:Attribute. ∀ j:Attribute. (j>i ⇒ columns[j] = N) ∧
3     (j≤i ⇒ columns[j] < r.len);
4
5 pred project_spec(s:Relation, r:Relation, columns:Array[N,Length]) ⇔
6 s.len = |{i | i:Attribute with columns[i] ≠ N}| ∧
7 ∀ tr:Row. tr∈r.tup ⇒ ∃ ts:Row. ts∈s.tup ∧
8     ∀ i:Attribute. i < s.len ⇒ ts[i]=tr[columns[i]];
```

Listing 3.19: Pre- and postconditions for π

The first part of the procedure resembles the implicit definition. At line 8 of Listing 3.20, a loop iterates over all of the tuples in *r*, and in another loop, the projection of a single row is created and added to the set of tuples of the output. Note that RISCAL allows us to use two kinds of for loops: one that functions as an iterator for a set and another one that iteratively increments an integer variable.

```
 1 proc project2(r:Relation, columns:Array[N,Length]):Relation
 2 requires project_prec(r, columns);
 3 ensures project_spec(result,r,columns); {
 4
 5     var l:Length := |{i | i:Attribute with columns[i] ≠ N}|;
 6     var q:Relation := ⟨len: l, tup: choose s:Set[Row] with |s|=0⟩;
 7
 8     for t ∈ r.tup do {
 9         var tn:Row := Array[N,Element](0);
10
11         var j:Length := 0;
12         for var i:Length := 0; i<N; i:=i+1 do {
13             if columns[i] ≠ N then {
14                 tn[j] := t[columns[i]];
15                 j := j+1;
16             }
17         }
18         q.tup := q.tup ∪ {tn};
19     }
20
21     return q;
22 }
```

Listing 3.20: Explicit implementation of π

For the formalization of a particular theorem and parts of Chapter 4 we will need an auxiliary function that takes a list of attributes and converts it into an array as it is necessary for the second argument of `project2`. For this, we utilize the recursive structure of the type `List`. While it is not an empty list, we remove layers and collect the attribute values in an array.

An important language construct is the keyword `match`. With it, we can recognize certain patterns of values of recursive types and map them to values of arbitrary other types. In this case, we use it to separate the attribute and the list in the interior of a node (see Listing 3.21).

```
proc attributes(att:List):Array[N,Length] {
    var arr:Array[N,Length] := Array[N,Length](N);

    var l:List := att;
    var i:Attribute := 0;

    while l ≠ List!nil do {
        arr[i] := match l with {
            node(a:Attribute, li:List) -> a;
        };
        l := match l with {
            node(a:Attribute, li:List) -> li;
        };
        i := i+1;
    }

    return arr;
}
```

Listing 3.21: Auxillary converter procedure

Theorems

Because there are no suitable literals or constructors for arrays in RISCAL, we depend on the recursive list from earlier and the function `parseProject`. Again the attributes must be in the range of the arity of the relation. Because we create a Cartesian product, the precondition must contain a bound for the cardinality of the Cartesian product of *r* with itself, which is `2*|r.tup|` (see Listing 3.22).

```
    theorem project_cartesian_subset(r:Relation, a:Attribute, b:Attribute)
    requires a < r.len ∧ b < r.len ∧ 2*|r.tup| ≤ M; ⇔
    project2(r,parseProject(List!node(a, List!node(b, List!nil)))).tup
    ⊆ cartesian2(project2(r, parseProject(List!node(a, List!nil))),
        project2(r, parseProject(List!node(b, List!nil)))).tup;
```

Listing 3.22: Implementation of Theorem 3.3

3.2.7. Join

The formalization of ⋈ strongly resembles that of *C* (see Listing 3.23). If one looks at the mathematical definition, this is no coincidence since we know that a join of two relations is a subset of their Cartesian product.

```
pred join_prec(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute) ⇔
n1<r1.len ∧ n2<r2.len ∧ r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;

pred join_spec(s:Relation, r1:Relation, r2:Relation, n1:Attribute, n2:
    Attribute) ⇔
s.len = r1.len+r2.len ∧ ∀ t:Row. t∈s.tup ⇔ ∃ t1:Row, t2:Row.
    (t1∈r1.tup ∧ t2∈r2.tup ∧ concat_spec(t,t1,t2,r1.len,r2.len) ∧ t1[n1] =
        t2[n2]);
```

Listing 3.23: Pre- and postconditions for ⋈

In the approach with the implicit set definition, we can spot the same similarity as in the last definition. In Listing 3.24, the addition, in this case, is the `with` clause.

```
fun join2(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute):Relation
requires join_prec(r1, r2, n1, n2);
ensures join_spec(result,r1,r2,n1,n2);
= ⟨len: r1.len+r2.len, tup: {concat(t1,t2,r1.len,r2.len) | t1∈r1.tup, t2∈r2
    .tup with t1[n1] = t2[n2]}⟩;
```

Listing 3.24: Explicit implementation of ⋈ (version 1)

The algorithmic description of ⋈ adds a conditional part to that of *C* that is shown in Listing 3.25.

```
proc join3(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute):Relation
requires join_prec(r1, r2, n1, n2);
ensures join_spec(result,r1,r2,n1,n2); {
    var q:Relation := ⟨len: r1.len+r2.len, tup: choose s:Set[Row] with |s
        |=0⟩;

    for t1 ∈ r1.tup do {
        for t2 ∈ r2.tup do {
            if t1[n1] = t2[n2] then {
                q.tup := q.tup ∪ {concat(t1, t2, r1.len, r2.len)};
            }
        }
    }

    return q;
}
```

Listing 3.25: Explicit implementation of ⋈ (version 2)

3.3. Model Checking Results

In the following Tables 3.1 and 3.2, we give the times required (in milliseconds) required by each SMT solver (Yices, Z3, Boolector, CVC4) to check the postconditions of operations or theorems follow. The parameters for the benchmark were $M = 2$ and $N = 2$. The system architecture on what the benchmark was run can be found in Appendix A.

	Yices	**Z3**	**Boolector**	**CVC4**	**MC**	**TP (Z3)**
rUnion	97	115	36	72	52	5473
rIntersect	49	23	40	59	41	5341
rMinus	37	22	8	39	45	5180
concat	12	67	1119	269	247	364
cartesian	54	1490	36095	26230	467	395
select	52	64	1210	661	92	323
project	43	122	27419	2965	97	560
join	88	3448	191882	68600	86	386

Table 3.1.: Benchmark Operations (ms)

	Yices	Z3	Boolector	CVC4	MC	TP (Z3)
select_union_equiv	7	53	30	107	112	289
select_intersect_equiv	31	54	9	85	187	285
select_minus_equiv	53	22	35	108	182	251
select_intersect_comp	30	38	38	98	67	294
join_cartesian_subset	67	4285	167761	107472	83	4109

Table 3.2.: Benchmark Theorems (ms)

For comparison, the respective times of RISCAL's model checker based on semantic evaluation (MC, parallel execution with 4 threads) and of the RISCTP theorem proving interface using Z3 as an external theorem prover (TP (Z3)) are also given. Yices behaves most reliably because the check terminates for all theorems here. RISCTP is also doing very well in time, especially with the set operations. However, the fastest results for checking theorems come from Z3.

4. Query Language

In this chapter, we provide the abstract syntax and denotational semantics of a query language similar to SQL. This language strongly resembles the language of relational algebra, and we will find a way to express its counterpart in real standardized SQL. All of this will be implemented in RISCAL.

4.1. Abstract Syntax

Most importantly, we need a syntactical description for each relational algebra operation. Therefore, the syntactical domain `Query` has an alternative for each operation. RISCAL does not support a variable argument list, but projection requires one. Using the domain `List`, we can avoid this problem.

Definition 4.1 *The language* $\texttt{SQL}_{min}$ *consists of all expressions in the syntactical domain* `Query`*. The natural variables* T *and* A *identify tables and attributes, whereas* E *denotes* 0 *or* 1.

$$\begin{array}{lcl}
L & \in & \textit{List} \\
Q & \in & \textit{Query} \\
L & ::= & \texttt{()} \mid \texttt{(}A\texttt{,}L\texttt{)} \\
Q & ::= & \texttt{from(}T\texttt{)} \\
 & \mid & \texttt{on(}A_1\texttt{,}A_2\texttt{,}Q_1\texttt{,}Q_2\texttt{)} \\
 & \mid & \texttt{where(}A\texttt{,}E\texttt{,}Q\texttt{)} \\
 & \mid & \texttt{select(}L\texttt{,}Q\texttt{)} \\
 & \mid & \texttt{add(}Q_1\texttt{,}Q_2\texttt{)} \\
 & \mid & \texttt{inters(}Q_1\texttt{,}Q_2\texttt{)} \\
 & \mid & \texttt{minus(}Q_1\texttt{,}Q_2\texttt{)} \\
 & \mid & \texttt{cart(}Q_1\texttt{,}Q_2\texttt{)}
\end{array}$$

The structure of the syntax already gives a hint as to its meaning. In the following section, we will connect `add`(Q_1, Q_2), which is a binary function symbol in the language, to the set operation $\cup$ (and similar for the other functions of the language).

4.2. Denotational Semantics

The method of denotational semantics requires that we establish a semantic domain. In our case, this is the domain of all relations `Relation` (see Definition 3.1) on arbitrary attribute schemas. Additionally, we define a collection of "base relations" in the form of a map db from table identifiers i to concrete relations on the attribute schema $\mathcal{A}_i$.

Definition 4.2 *Let* db : $\texttt{TableId} \to \bigcup_{i=1}^{n} \texttt{Relation}_{\mathcal{A}_i}$ *be a database. We define the denotational semantics* $[\![\cdot]\!]_{\mathrm{db}}$: *Query* $\to$ `Relation` *of* $\texttt{SQL}_{min}$ *by structural induction in the following way:*

$$\begin{aligned}
[\![\texttt{from}(T)]\!]_{\mathrm{db}} &:= \mathrm{db}(T)\\
[\![\texttt{on}(A_1, A_2, Q_1, Q_2)]\!]_{\mathrm{db}} &:= \bowtie([\![Q_1]\!]_{\mathrm{db}}, [\![Q_2]\!]_{\mathrm{db}}, A_1, A_2)\\
[\![\texttt{where}(A, E, Q)]\!]_{\mathrm{db}} &:= \sigma([\![Q]\!]_{\mathrm{db}}, A, E)\\
[\![\texttt{select}(L, Q)]\!]_{\mathrm{db}} &:= \pi([\![Q]\!]_{\mathrm{db}}, L)\\
[\![\texttt{add}(Q_1, Q_2)]\!]_{\mathrm{db}} &:= [\![Q_1]\!]_{\mathrm{db}} \cup [\![Q_2]\!]_{\mathrm{db}}\\
[\![\texttt{inters}(Q_1, Q_2)]\!]_{\mathrm{db}} &:= [\![Q_1]\!]_{\mathrm{db}} \cap [\![Q_2]\!]_{\mathrm{db}}\\
[\![\texttt{minus}(Q_1, Q_2)]\!]_{\mathrm{db}} &:= [\![Q_1]\!]_{\mathrm{db}} \setminus [\![Q_2]\!]_{\mathrm{db}}\\
[\![\texttt{cart}(Q_1, Q_2)]\!]_{\mathrm{db}} &:= C([\![Q_1]\!]_{\mathrm{db}}, [\![Q_2]\!]_{\mathrm{db}})
\end{aligned}$$

For the operations $\bowtie$, σ, π and C see Definitions 3.3 to 3.6.

4.3. RISCAL Model

Like earlier, for the auxiliary type `List`, we use the recursive types of RISCAL. Listing 4.1 starts with an alternative denoting a minimal query, i.e., with the query `from`, by which a database table can be retrieved. Mainly, the listing represents an implementation of Definition 4.1.

```
rectype(D) Query = from(TableId)
| on(Attribute, Attribute, Query, Query)
| where(Attribute, Element, Query)
| select(List, Query)
| add(Query, Query)
| inters(Query, Query)
| minus(Query, Query)
| cart(Query, Query);
```

Listing 4.1: Recursive datatype `Query`

Listing 4.2 implements Definition 4.2, which can be entered with only minor adjustments due to the RISCAL keyword `match`. It should be noted that `db` is a parameter of the `query` function.

```
proc query(db:Database, q:Query):Relation {
    var r:Relation := match q with {
        from(tid:TableId) -> db[tid];
        on(n1:Attribute, n2:Attribute, q1:Query, q2:Query) -> join(query(db
            , q1), query(db, q2), n1, n2);
        where(a:Attribute, e:Element, q:Query) -> select(query(db, q), a, e
            );
        select(a:List, q:Query) -> project(query(db, q), attributes(a));
        add(q1:Query, q2:Query) -> rUnion(query(db, q1), query(db, q2));
        inters(q1:Query, q2:Query) -> rIntersect(query(db, q1), query(db,
            q2));
        minus(q1:Query, q2:Query) -> rMinus(query(db, q1), query(db, q2));
        cart(q1:Query, q2:Query) -> cartesian(query(db, q1), query(db, q2))
            ;
    };
    return r;
}
```

Listing 4.2: Procedure query

In the following, we describe how an expression in "Mini-SQL" can be translated into regular SQL. A SQL query is always executed in the same order, which means we can sort the clauses in the following way:

$$\underset{4}{\texttt{FROM}} \succ \underset{3}{\texttt{ON}} \succ \underset{2}{\texttt{WHERE}} \succ \underset{1}{\texttt{SELECT}},$$

where $C_1 \succ C_2$ means that clause C_1 is evaluated before clause C_2. If the corresponding operations of the "Mini-SQL" are composed in this order, they are equivalent to a single SQL query without subqueries. We can list every possibility because:

- The leaves of any AST must contain `FROM` clauses, and
- Under the assumption that the tree nodes of an AST contain clauses of strictly ascending priority, there are a maximum of 4 layers.

This gives us eight "atomic" SQL expressions listed below.

$$\begin{tikzcd} & & & 4 \\ 1 \arrow[r, dash] & 2 \arrow[r, dash] & 3 \arrow[ur, dash] \arrow[dr, dash] & \\ & & & 4 \end{tikzcd}$$

```
SELECT <attributes>
FROM <table>
JOIN <table>
ON <attribute> = <attribute>
WHERE <attribute> = <value>;
```

$$\begin{tikzcd} & & 4 \\ 2 \arrow[r, dash] & 3 \arrow[ur, dash] \arrow[dr, dash] & \\ & & 4 \end{tikzcd}$$

```
SELECT *
FROM <table>
JOIN <table>
ON <attribute> = <attribute>
WHERE <attribute> = <value>;
```

$$\begin{tikzcd} & & 4 \\ 1 \arrow[r, dash] & 3 \arrow[ur, dash] \arrow[dr, dash] & \\ & & 4 \end{tikzcd}$$

```
SELECT <attributes>
FROM <table>
JOIN <table>
ON <attribute>
    = <attribute>;
```

$$\begin{tikzcd} 1 \arrow[r, dash] & 2 \arrow[r, dash] & 4 \end{tikzcd}$$

```
SELECT <attributes>
FROM <table>
WHERE <attribute> = <value>;
```

$$\begin{tikzcd} & 4 \\ 3 \arrow[ur, dash] \arrow[dr, dash] & \\ & 4 \end{tikzcd}$$

```
SELECT *
FROM <table>
JOIN <table>
ON <attribute> = <attribute>;
```

$$\begin{tikzcd} 2 \arrow[r, dash] & 4 \end{tikzcd}$$

```
SELECT *
FROM <table>
WHERE <attribute>
= <value>;
```

$$\begin{tikzcd} 1 \arrow[r, dash] & 4 \end{tikzcd}$$

```
SELECT <attributes>
FROM <table>;
```

$$4$$

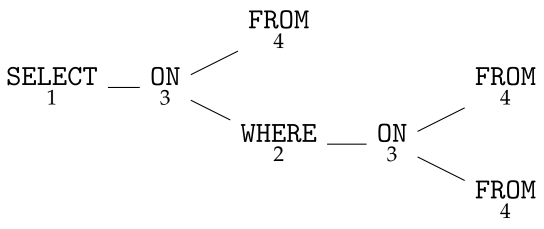
```
SELECT *
FROM <table>;
```

Of course, more queries can be carried out by the "Mini-SQL", but with a little effort, they can also be translated. Take, for example, the query:

```
Query!select(1, Query!on(6, 3, Query!where(3, 0, Query!on(2, 2, Query!from
    (0), Query!from(1))), Query!from(2)));
```

Listing 4.3: Example query in RISCAL

If we write the AST of this query while ignoring all arguments that are not of the type `Query`, we get the following picture.



The edge between the first `ON` node and the `WHERE` node is a case where the priority of the operations is not strictly ascending. This indicates that everything above this edge is the outer query, and everything below this edge is the inner query. In particular, the corresponding SQL query is given by the listing below. The subquery is made visible through indentation.

```
SELECT 1, 2
FROM (
    SELECT *
    FROM 0
    JOIN 1
    ON 2 = 2
    WHERE 3 = 0
)
JOIN 2
ON 6 = 3;
```

Listing 4.4: Example query in SQL

More generally, consider an AST of a "Mini-SQL" expression, s.t. there exists an edge (C_1, C_2) with $C_1 \succeq C_2$, i.e., the node nearer to the root has higher or equal priority than the other node. If C_1 is either equal to SELECT or WHERE, then the subquery will be placed after the FROM keyword of the SQL query. Note that this is just a keyword and has nothing to do with the RISCAL function from. If C_1 is equal to ON, then it depends on the position C_2. If C_2 is the first child of C_1, then the subquery is placed after the FROM keyword, else after the JOIN keyword.

```
proc query_to_sql(q0:Query):() {
    match q0 with {
        add(q1_1:Query, q1_2:Query) -> {
            translate_add(q1_1, q1_2);
        }
        inters(q1_1:Query, q1_2:Query) -> {
            translate_inters(q1_1, q1_2);
        }
        minus(q1_1:Query, q1_2:Query) -> {
            print "Not implemented";
        }
        cart(q1_1:Query, q1_2:Query) -> {
            translate_cart(q1_1, q1_2);
        }
        select(l:List, q1:Query) -> {
            translate_select(l, q1);
        }
        where(a:Attribute, e:Element, q1:Query) -> {
            print "SELECT *";
            translate_where(a, e, q1);
        }
        on(n1:Attribute, n2:Attribute, q1_1:Query, q1_2:Query) -> {
            print "SELECT *";
            translate_on(n1, n2, q1_1, q1_2);
        }
        from(tid:TableId) -> {
            print "SELECT *";
            translate_from(tid);
        }
    }
}
```

Listing 4.5: Implementation of query_to_sql

Listing 4.5 describes the procedure for printing an equivalent SQL statement for a given query. The expression `q0` is matched based on the root of its AST. If the root is not `SELECT`, there is no projection, and the statement will start with `SELECT *` to select all attributes of the relation. The procedures starting with `translate` hide recursive calls to `query_to_sql`.

```
proc translate_add(q0_1:Query, q0_2:Query):() {
    print "(";
    query_to_sql(q0_1);
    print ") UNION";
    print "(";
    query_to_sql(q0_2);
    print ")";
}

proc translate_inters(q0_1:Query, q0_2:Query):() {
    print "(";
    query_to_sql(q0_1);
    print ") INTERSECT";
    print "(";
    query_to_sql(q0_2);
    print ")";
}
```

Listing 4.6: Implementation of translation procedures for the set operations

The two set operations, union and intersection, are binary. The left-hand side and right-hand side queries are translated and concatenated with one of the SQL keywords `UNION` or `INTERSECT`, as shown in Listing 4.6.

```
proc translate_select(l:List, q0:Query):() {
    var att:List := l;
    print "SELECT";
    while att ≠ List!nil do {
        var a:Attribute := match att with {
            node(a:Attribute, li:List) -> a;
        };
        att := match att with {
            node(a:Attribute, li:List) -> li;
        };
        if att ≠ List!nil then {
```

```
12              print "{1}, ", a;
13          } else {
14              print a;
15          }
16      }
17      match q0 with {
18          where(a:Attribute, e:Element, q1:Query) -> {
19              translate_where(a, e, q1);
20          }
21          on(n1:Attribute, n2:Attribute, q1_1:Query, q1_2:Query) -> {
22              translate_on(n1, n2, q1_1, q1_2);
23          }
24          from(tid:TableId) -> {
25              translate_from(tid);
26          }
27          _ -> {
28              translate_blank(q0);
29          }
30      }
31  }
```

Listing 4.7: Implementation of `translate_select`

Listing 4.7 presents a procedure that takes a list of attributes and a query and prints the SQL equivalent of the query on the given attributes. The procedure first loops over all attributes by deconstructing every list node into the contained attribute and the inner list (see lines 5 and 8) and prints them separated by commas.

The query `q0` is matched according to its root. If its priority is higher than the priority of `SELECT`, then the output is enhanced (the query remains atomic). A subquery is produced if the root is of the same priority, i.e., the one of `SELECT`. The subquery occurs after the SQL keyword `FROM`, as shown in Listing 4.8. The process of producing subqueries follows the same paradigm across all translation procedures.

```
proc translate_blank(q0:Query):() {
    print "FROM (";
    query_to_sql(q0);
    print ")";
}
```

Listing 4.8: Implementation of `translate_blank`

Listing 4.9 shows the procedure `translate_from`, which is similar to `translate_blank` in that it also prints a `FROM` clause. However, while `translate_blank` creates a subquery, `translate_from` prints the reference to the specified table.

```
proc translate_from(tid:TableId):() {
print "FROM {1}", tid;
}
```

Listing 4.9: Implementation of `translate_from`

The procedure `translate_where` (Listing 4.10) takes an attribute and an element of the domain, and a query and adds a `WHERE` clause at the end. Once again, the root node of the AST of `q0` is considered. The two clauses with higher priority are `ON` and `FROM`, handled on lines 3 and 6, respectively. If neither of these clauses is present, a subquery is produced in line 9.

```
1  proc translate_where(a:Attribute, e:Element, q0:Query):() {
2      match q0 with {
3          on(n1:Attribute, n2:Attribute, q1_1:Query, q1_2:Query) -> {
4              translate_on(n1, n2, q1_1, q1_2);
5          }
6          from(tid:TableId) -> {
7              translate_from(tid);
8          }
9          _ -> {
10             translate_blank(q0);
11         }
12     }
13     print "WHERE {1} = {2}", a, e;
14 }
```

Listing 4.10: Implementation of `translate_where`

Listing 4.11 presents the procedure `translate`$_c$`art`, corresponding to a binary operation. This procedure first checks if the root nodes of the input query ASTs are `FROM` (using pattern matching in lines 2 and 6). If so, it extracts the identifiers of the tables. Otherwise, it returns a tuple with `false` as the first component and a placeholder identifier `0` as the second component.

Since there is no projection, the first line printed is `SELECT *` (line 10). In line 11, a nested If-Then-Else statement is used to construct the `FROM` clause based on the structure of the input queries. Depending on whether the queries produce subqueries or table identifiers after the `FROM` clause, none, one, or two subqueries are printed.

```
1  proc translate_cart(q0_1:Query, q0_2:Query):() {
2      var q1_term:Tuple[Bool, TableId] := match q0_1 with {
3          from(tid:TableId) -> ⟨true, tid⟩;
4          _ -> ⟨false, 0⟩;
5      };
6      var q2_term:Tuple[Bool, TableId] := match q0_2 with {
7          from(tid:TableId) -> ⟨true, tid⟩;
8          _ -> ⟨false, 0⟩;
9      };
10     print "SELECT *";
11     if q1_term.1 then {
12         if q2_term.1 then {
13             print "FROM {1}, {2}", q1_term.2, q2_term.2;
14         } else {
15             print "FROM {1}, (", q1_term.2;
16             query_to_sql(q0_2);
17             print ")";
18         }
19     } else {
20         if q2_term.1 then {
21             print "FROM (";
22             query_to_sql(q0_1);
23             print "), {1}", q2_term.2;
24         } else {
25             print "FROM (";
26             query_to_sql(q0_1);
27             print "), (";
28             query_to_sql(q0_2);
29             print ")";
30         }
31     }
32 }
```

Listing 4.11: Implementation of `translate_cart`

Listing 4.12 presents the procedure `translate_on`, which is similar to `translate_cart` (Listing 4.11), but with a different output format. Instead of producing a comma-separated list of subqueries or table identifiers, `translate_on` produces two separate clauses for the left-hand and right-hand side queries. The `FROM` clause contains only the left-hand side query, and the `JOIN` clause introduces the right-hand side query. At line 36, the procedure prints the `ON` clause that specifies the join condition.

```
 1 proc translate_on(n1:Attribute, n2:Attribute, q0_1:Query, q0_2:Query):() {
 2     var q1_term:Tuple[Bool, TableId] := match q0_1 with {
 3         from(tid:TableId) -> ⟨true, tid⟩;
 4         _ -> ⟨false, 0⟩;
 5     };
 6     var q2_term:Tuple[Bool, TableId] := match q0_2 with {
 7         from(tid:TableId) -> ⟨true, tid⟩;
 8         _ -> ⟨false, 0⟩;
 9     };
10
11     if q1_term.1 then {
12         if q2_term.1 then {
13             print "FROM {1}", q1_term.2;
14             print "JOIN {1}", q2_term.2;
15         } else {
16             print "FROM {1}", q1_term.2;
17             print "JOIN (";
18             query_to_sql(q0_2);
19             print ")";
20         }
21     } else {
22         if q2_term.1 then {
23             print "FROM (";
24             query_to_sql(q0_1);
25             print ")";
26             print "JOIN {1}", q2_term.2;
27         } else {
28             print "FROM (";
29             query_to_sql(q0_1);
30             print ")";
31             print "JOIN (";
32             query_to_sql(q0_2);
33             print ")";
34         }
```

```
    }
    print "ON {1} = {2}", n1, n2;
}
```

Listing 4.12: Implementation of `translate_on`

4.4. Executions

In order to verify the correctness of the work presented in this chapter, we will use a simple database as a basis for our test cases. The general structure of the database can be seen in Table 4.1.

0	
0	**1**
0	0
0	1

1	
0	**1**
0	1
1	0
1	1

Table 4.1.: Sample database

The sample database can be created in Sqlite using the DDL script shown in Listing 4.13.

```
BEGIN TRANSACTION;
CREATE TABLE IF NOT EXISTS "0" (
  "0" INTEGER,
  "1" INTEGER
);
CREATE TABLE IF NOT EXISTS "1" (
  "0" INTEGER,
  "1" INTEGER
);
INSERT INTO "0" ("0","1") VALUES (0,0);
INSERT INTO "0" ("0","1") VALUES (0,1);
INSERT INTO "1" ("0","1") VALUES (0,1);
INSERT INTO "1" ("0","1") VALUES (1,0);
INSERT INTO "1" ("0","1") VALUES (1,1);
COMMIT;
```

Listing 4.13: DDL script for sample database

We also create a RISCAL procedure that returns the database as an array of relations (Listing 4.14).

```
proc sample_database():Array[K, Relation] {
   var dum:Map[Attribute,Element] := Map[Attribute,Element](0);

   var emp:Relation := ⟨len: 0, tup: choose s:Set[Row] with |s|=0⟩;
   var r1:Relation := ⟨len: 2, tup: choose s:Set[Row] with |s|=0⟩;
   var r2:Relation := ⟨len: 2, tup: choose s:Set[Row] with |s|=0⟩;

   r1.tup := r1.tup ∪ {dum};
   dum[1] := 1;
   r1.tup := r1.tup ∪ {dum};
   r2.tup := r2.tup ∪ {dum};
   dum[0] := 1;
   r2.tup := r2.tup ∪ {dum};
   dum[1] := 0;
   r2.tup := r2.tup ∪ {dum};

   var db:Array[K, Relation] := Array[K, Relation](emp) with [0]=r1 with
       [1]=r2;

```

```
    return db;
}
```

Listing 4.14: RISCAL procedure for sample database

Each test case in RISCAL is associated with a query q. The following steps are followed for each test case:

1. Execute the query q using the `query` procedure in RISCAL.
2. Translate q to SQL and make necessary adaptations for the Sqlite system.
3. Execute the query in Sqlite.

The test case is considered successful if the output of RISCAL and Sqlite matches.

4.4.1. Atomic Queries

Test Cases

The first query (see Listing 4.15) has every possible clause of an atomic query.

```
var q1:Query := Query!select(List!node(2, List!node(0, List!nil)), Query!
    where(1, 0, Query!on(0, 1, Query!from(0), Query!from(1))));
```

Listing 4.15: Syntactic query in RISCAL

Executing `query_to_sql(q1)` in RISCAL produces the output shown in Listing 4.16.

```
SELECT
2,
0
FROM 0
JOIN 1
ON 0 = 1
WHERE 1 = 0
```

Listing 4.16: Translation into SQL

To make the output compatible with SQLite, some changes need to be made:

- Tables and column names must be enclosed in quotes.
- The table's name containing the column must be written before the column name, separated by a dot.
- Since `SELECT` $\succ$ `JOIN`, the projection on attribute with index 2 refers to the first attribute of the right-hand table of the join. Therefore, the index must be replaced with `"1"."0"`, where `1` is the second table and `0` is its first attribute.

Listing 4.17 shows the query executed in SQLite and its result.

```
sqlite> SELECT "1"."0", "0"."0"
...> FROM "0"
...> JOIN "1"
...> ON "0"."0" = "1"."1"
...> WHERE "0"."1" = 0;
1|0
```

Listing 4.17: Execution in SQLite

When we execute `query`(`db`, `q1`) in RISCAL, the system outputs [`2`, {[`1`, `0`, `0`, `0`]}] as the result. This matches the output obtained from SQLite. Other test cases can be found in Appendix A. Their resulting outputs can be found in the table below.

Query	RISCAL	SQLite

`Query!select(1,`
`Query!where(1, 1,`
`Query!from(0)))`
	`Query result:`
	`[2,{[1,0,0,0]}]`
	`SELECT`
	`1,`
	`0`
	`FROM 0`
	`WHERE 1 = 1`
	`;`
		`sqlite> SELECT "0"."1", "0"."0"`
		`   ...> FROM "0"`
		`   ...> WHERE "0"."1" = 1;`
		`1|0`

`Query!select(`
`List!node(1,`
`List!node(2,`
`List!nil)),`
`Query!on(0, 1,`
`Query!from(0),`
`Query!from(1)));`
	`Query result:`
	`[2,{[0,1,0,0],[1,1,0,0]}]`
	`SELECT`
	`1,`
	`2`
	`FROM 0`
	`JOIN 1`
	`ON 0 = 1`
	`;`
		`sqlite> SELECT "0"."1", "1"."0"`
		`   ...> FROM "0"`
		`   ...> JOIN "1"`
		`   ...> ON "0"."0" = "1"."1";`
		`0|1`
		`1|1`

`Query!select(1,`
`Query!from(1));`
	`Query result:`
	`[2,{[1,0,0,0],[1,1,0,0],`
	`[0,1,0,0]}]`
	`SELECT`
	`1,`
	`0`
	`FROM 1`
	`;`
		`sqlite> SELECT "1"."1", "1"."0"`
		`   ...> FROM "1";`
		`1|0`
		`0|1`
		`1|1`

`Query!where(1, 0,`
`Query!on(0, 1,`
`Query!from(0),`
`Query!from(1)));`
	`Query result:`
	`[4,{[0,0,1,0]}]`
	`SELECT *`
	`FROM 0`
	`JOIN 1`
	`ON 0 = 1`
	`WHERE 1 = 0`
	`;`
		`sqlite> SELECT *`
		`   ...> FROM "0"`
		`   ...> JOIN "1"`
		`   ...> ON "0"."0" = "1"."1"`
		`   ...> WHERE "0"."1" = "0"."0";`
		`0|0|1|0`

Query	RISCAL	SQLite

`Query!where(1, 0,`
`Query!from(0));`
	`Query result:`
	`[2,{[0,0,0,0]}]`
	`SELECT *`
	`FROM 0`
	`WHERE 1 = 0`
	`;`
		`sqlite> SELECT *`
		`   ...> FROM "0"`
		`   ...> WHERE "0"."1" = "0"."0";`
		`0|0`

`Query!on(0, 1,`
`Query!from(0),`
`Query!from(1));`
	`Query result:`
	`[4,{[0,0,1,0],[0,1,1,0]}]`
	`SELECT *`
	`FROM 0`
	`JOIN 1`
	`ON 0 = 1`
	`;`
		`sqlite> SELECT *`
		`   ...> FROM "0"`
		`   ...> JOIN "1"`
		`   ...> ON "0"."0" = "1"."1";`
		`0|0|1|0`
		`0|1|1|0`

`Query!from(0);`	`Query result:`
	`[2,{[0,0,0,0],[0,1,0,0]}]`
	`SELECT *`
	`FROM 0`
	`;`
		`sqlite> SELECT *`
		`   ...> FROM "0";`
		`0|0`
		`0|1`

4.4.2. Non-atomic Queries

We have already tested the translation and execution of atomic queries and are now proceeding with non-atomic queries. As an example, consider the query shown in Listing 4.18.

```
  var qq:Query := Query!where(2, 0, Query!on(0, 1, Query!from(0), Query!
      select(List!node(1, List!node(2, List!nil)), Query!on(0, 0, Query!
      from(0), Query!from(1)))));
```

Listing 4.18: Syntactic query in RISCAL

When we translate it using the RISCAL procedure, we obtain the query shown in Listing 4.19.

```
SELECT *
FROM 0
JOIN (
SELECT
1,
2
FROM 0
JOIN 1
ON 0 = 0
)
ON 0 = 1
WHERE 2 = 0
;
```

Listing 4.19: Translation into SQL

In the previous section, we saw how to adapt an atomic query to meet the requirements of Sqlite. The same approach can be applied to subqueries. However, for the outer query, we need to consider additional aspects. Firstly, every subquery must be given a name for reference. In this case, we name it `_1`. Secondly, we need to rename every attribute in the enclosing query following the order of clauses. The resulting query is shown in Listing 4.20.

```
sqlite> SELECT *
   ...> FROM "0"
   ...> JOIN (
   ...> SELECT "0"."1", "1"."0"
   ...> FROM "0"
   ...> JOIN "1"
   ...> ON "0"."0" = "1"."0"
   ...> ) AS "_1"
   ...> ON "0"."0" = "_1"."1"
   ...> WHERE "_1"."0" = 0;
0|0|0|0
0|1|0|0
```

Listing 4.20: Execution in SQLite

Executing this query in RISCAL produces $[4, \{[0,0,0,0],[0,1,0,0]\}]$ as output, which matches the result obtained from SQLite earlier. Again, other test cases can be found in the table below, and their implementation is given in Appendix A.

Query	**RISCAL**	**SQLite**

`Query!on(1, 0,`
`Query!select(`
`List!node(1,`
`List!node(0,`
`List!nil)),`
`Query!from(1)),`
`Query!select(`
`List!node(0,`
`List!nil),`
`Query!from(0)));`
	`Query result:`
	`[3,{[1,0,0,0]}]`
	`SELECT *`
	`FROM (`
	`SELECT`
	`1,`
	`0`
	`FROM 1`
	`)`
	`JOIN (`
	`SELECT`
	`0`
	`FROM 0`
	`)`
	`ON 1 = 0`
	`;`
		`sqlite> SELECT *`
		`...> FROM (`
		`...> SELECT "1"."1" AS "0", "1"."0" AS "1"`
		`...> FROM "1"`
		`...> ) AS "_1"`
		`...> JOIN (`
		`...> SELECT "0"."0" AS "0"`
		`...> FROM "0"`
		`...> ) AS "_0"`
		`...> ON "_1"."1" = "_0"."0";`
		`1|0|0`
		`1|0|0`

`Query!where(1, 0,`
`Query!select(`
`List!node(1,`
`List!node(0,`
`List!nil)),`
`Query!from(1)));`
	`Query result:`
	`[2,{[1,0,0,0]}]`
	`SELECT *`
	`FROM (`
	`SELECT`
	`1,`
	`0`
	`FROM 1`
	`)`
	`WHERE 1 = 0`
	`;`
		`sqlite> SELECT *`
		`   ...> FROM (`
		`   ...> SELECT "1"."1" AS "0", "1"."0" AS "1"`
		`   ...> FROM "1"`
		`   ...> ) AS "_1"`
		`   ...> WHERE "_1"."1" = 0;`
		`1|0`

5. Conclusions and Further Work

In summary, we looked at the foundations of relational databases, namely relational algebra, and created a formalization accessible to model checkers, i.e., for validating the specification. This is very helpful because we could rely on correctly implemented operations to develop a query language similar to SQL.

This simple language had the advantage that it could be easily translated into standard SQL, allowing it to be tested not only in the RISCAL environment but also in SQLite, a well-known relational database. Nevertheless, this language qualifies as a tool to understand relational algebra as it should work without the peculiarities of a concrete SQL dialect. One could proceed by implementing this language in a general-purpose programming language. Its expressiveness is probably worse than that of real-world SQL implementations, but it can be assumed that its foundation conforms to a mathematically exact specification. This may be useful in the future.

We used all the different methods RISCAL offers us for model checking, with its semantic evaluation, SMT-LIB translation, and its latest addition - theorem proving via RISCTP. We mentioned that RISCTP works on top of external theorem provers, although a new backend with an internal solver compatible with first-order logic is planned. It would also have been interesting to see how well such a tool could handle the formal specification in this thesis, especially given the benchmark result.

A final remark concerns the generation of the specification itself. This task cannot yet be automated, but tools such as RISCAL, which acts as a sandbox for testing, can simplify this manual process and make it fun for mathematicians and programmers due to its ease of use.

A. RISCAL Source Code

```
val M:ℕ; // maximum cardinality of relations
val N:ℕ; // maximum length of rows/tuples
val K:ℕ; // maximum number of tables
val D:ℕ; // maximum query depth
val L:ℕ; // maximum list length

type Element = ℕ[1];
type Attribute = ℕ[N-1];
type Length = ℕ[N];
type TableId = ℕ[K];
type Row = Map[Attribute, Element];
type Relation = Record[len:Length, tup:Set[Row]]
with |value.tup| ≤ M ∧ ∀ t:Row, i:Attribute. t ∈ value.tup ∧ i ≥ value.len
    ⇒ t[i] = 0;
type Database = Array[K, Relation];

//////////////////////////////////////////////

pred union_compatible(r1:Relation, r2:Relation) ⇔ r1.len=r2.len;

fun rUnion(r1:Relation, r2:Relation):Relation
requires union_compatible(r1,r2) ∧ |r1.tup| + |r2.tup| ≤ M;
= ⟨len: r1.len, tup: r1.tup ∪ r2.tup⟩;

fun rIntersect(r1:Relation, r2:Relation):Relation
requires union_compatible(r1,r2);
= ⟨len: r1.len, tup: r1.tup ∩ r2.tup⟩;

fun rMinus(r1:Relation, r2:Relation):Relation
requires union_compatible(r1,r2);
= ⟨len: r1.len, tup: r1.tup \ r2.tup⟩;

//////////////////////////////////////////////
```

```

pred concat_prec(n1:Length, n2:Length) ⇔
n1 + n2 ≤ N;

pred concat_spec(t:Row, t1:Row, t2:Row, n1:Length, n2:Length) ⇔
∀ i:Attribute. (
if i < n1 then t[i] = t1[i]
else if i ≥ n1 ∧ i < n1+n2 then t[i] = t2[i-n1]
else t[i] = 0
);

fun concat1(t1:Row, t2:Row, n1:Length, n2:Length):Row
requires concat_prec(n1, n2);
= choose t:Row with concat_spec(t,t1,t2,n1,n2);

proc concat2(t1:Row, t2:Row, n1:Length, n2:Length):Row
requires concat_prec(n1, n2);
ensures concat_spec(result,t1,t2,n1,n2); {
  var t:Row := Array[N,Element](0);
  for var i:Length:=0; i<n1; i:=i+1 do {
    t[i] := t1[i];
  }
  for var i:Length:=n1; i<n1+n2; i:=i+1 do {
    t[i] := t2[i-n1];
  }
  return t;
}

fun concat(t1:Row, t2:Row, n1:Length, n2:Length):Row
requires concat_prec(n1, n2);
ensures concat_spec(result,t1,t2,n1,n2);
= concat2(t1, t2, n1, n2);

//////////////////////////////////////////////

pred cartesian_prec(r1:Relation, r2:Relation) ⇔
r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;

pred cartesian_spec(r:Relation, r1:Relation, r2:Relation) ⇔
r.len = r1.len+r2.len ∧
∀ t:Row. t∈r.tup ⇔ ∃ t1:Row, t2:Row.
  t1∈r1.tup ∧ t2∈r2.tup ∧ concat_spec(t,t1,t2,r1.len,r2.len);
```

```

fun cartesian1(r1:Relation, r2:Relation):Relation
requires cartesian_prec(r1, r2);
= choose r:Relation with cartesian_spec(r,r1,r2);

fun cartesian2(r1:Relation, r2:Relation):Relation
requires cartesian_prec(r1, r2);
ensures cartesian_spec(result,r1,r2);
= ⟨len: r1.len+r2.len, tup: {concat(t1,t2,r1.len,r2.len) | t1∈r1.tup, t2∈r2
    .tup}⟩;

proc cartesian3(r1:Relation, r2:Relation):Relation
requires cartesian_prec(r1, r2);
ensures cartesian_spec(result,r1,r2); {
  var q:Relation := ⟨len: r1.len+r2.len, tup: choose s:Set[Row] with |s|=0⟩
      ;

  for t1 ∈ r1.tup do {
    for t2 ∈ r2.tup do {
      q.tup := q.tup ∪ {concat(t1, t2, r1.len, r2.len)};
    }
  }

  return q;
}

fun cartesian(r1:Relation, r2:Relation):Relation
requires cartesian_prec(r1, r2);
ensures cartesian_spec(result,r1,r2);
= cartesian3(r1, r2);

//////////////////////////////////////////////

pred select_prec(r:Relation, a:Attribute) ⇔
a < r.len;

pred select_spec(s:Relation, r:Relation, a:Attribute, e:Element) ⇔
s.len = r.len ∧ ∀ t:Row. t∈s.tup ⇔ t∈r.tup ∧ t[a] = e;

fun select1(r:Relation, a:Attribute, e:Element):Relation
requires select_prec(r,a);
= choose s:Relation with select_spec(s,r,a,e);
```

```

fun select2(r:Relation, a:Attribute, e:Element):Relation
requires select_prec(r,a);
ensures select_spec(result,r,a,e);
= ⟨len: r.len, tup: {t | t∈r.tup with t[a] = e}⟩;

proc select3(r:Relation, a:Attribute, e:Element):Relation
requires select_prec(r,a);
ensures select_spec(result,r,a,e); {
  var q:Relation := ⟨len: r.len, tup: choose s:Set[Row] with |s|=0⟩;

  for t ∈ r.tup do {
    if t[a] = e then {
      q.tup = q.tup ∪ {t};
    }
  }
  return q;
}

fun select(r:Relation, a:Attribute, e:Element):Relation
requires select_prec(r,a);
ensures select_spec(result,r,a,e);
= select3(r, a, e);

///////////////////////////////////////////////

theorem select_union_equiv(r1:Relation, r2:Relation, a:Attribute, e:Element
    )
requires a < r1.len ∧ a < r2.len ∧ union_compatible(r1,r2) ∧ |r1.tup| + |r2
    .tup| ≤ M; ⇔
select2(rUnion(r1,r2),a,e) = rUnion(select2(r1, a, e), select2(r2, a, e));

theorem select_intersect_equiv(r1:Relation, r2:Relation, a:Attribute, e:
    Element)
requires a < r1.len ∧ a < r2.len ∧ union_compatible(r1,r2); ⇔
select2(rIntersect(r1,r2),a,e) = rIntersect(select2(r1, a, e), select2(r2,
    a, e));

theorem select_minus_equiv(r1:Relation, r2:Relation, a:Attribute, e:Element
    )
requires a < r1.len ∧ a < r2.len ∧ union_compatible(r1,r2); ⇔
select2(rMinus(r1,r2),a,e) = rMinus(select2(r1, a, e), select2(r2, a, e));
```

```

theorem select_intersect_comp(r:Relation, a:Attribute, e:Element, b:
    Attribute, f:Element)
requires a < r.len ∧ b < r.len; ⇔
select2(select2(r, a, e), b, f) = rIntersect(select2(r, a, e), select2(r, b
    , f));

//////////////////////////////////////////////

pred project_prec(r:Relation, columns:Array[N,Length]) ⇔
∃ i:Attribute. ∀ j:Attribute. (j>i ⇒ columns[j] = N) ∧
  (j≤i ⇒ columns[j] < r.len);

pred project_spec(s:Relation, r:Relation, columns:Array[N,Length]) ⇔
s.len = |{i | i:Attribute with columns[i] ≠ N}| ∧
∀ tr:Row. tr∈r.tup ⇒ ∃ ts:Row. ts∈s.tup ∧
  ∀ i:Attribute. i < s.len ⇒ ts[i]=tr[columns[i]];

fun project1(r:Relation, columns:Array[N,Length]):Relation
requires project_prec(r, columns);
= choose s:Relation with project_spec(s,r,columns);

proc project2(r:Relation, columns:Array[N,Length]):Relation
requires project_prec(r, columns);
ensures project_spec(result,r,columns); {

  var l:Length := |{i | i:Attribute with columns[i] ≠ N}|;
  var q:Relation := ⟨len: l, tup: choose s:Set[Row] with |s|=0⟩;

  for t ∈ r.tup do {
    var tn:Row := Array[N,Element](0);

    var j:Length := 0;
    for var i:Length := 0; i<N; i:=i+1 do {
      if columns[i] ≠ N then {
        tn[j] := t[columns[i]];
        j := j+1;
      }
    }
    q.tup := q.tup ∪ {tn};
  }

  return q;
}

fun project(r:Relation, columns:Array[N,Length]):Relation
requires project_prec(r, columns);
ensures project_spec(result,r,columns);
= project2(r, columns);

/////////////////////////////////////////////

pred join_prec(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute) ⇔
n1<r1.len ∧ n2<r2.len ∧ r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;

pred join_spec(s:Relation, r1:Relation, r2:Relation, n1:Attribute, n2:
    Attribute) ⇔
s.len = r1.len+r2.len ∧ ∀ t:Row. t∈s.tup ⇔ ∃ t1:Row, t2:Row.
  (t1∈r1.tup ∧ t2∈r2.tup ∧ concat_spec(t,t1,t2,r1.len,r2.len) ∧ t1[n1] = t2
      [n2]);

fun join1(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute):Relation
requires join_prec(r1, r2, n1, n2);
= choose s:Relation with join_spec(s,r1,r2,n1,n2);

fun join2(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute):Relation
requires join_prec(r1, r2, n1, n2);
ensures join_spec(result,r1,r2,n1,n2);
= ⟨len: r1.len+r2.len, tup: {concat(t1,t2,r1.len,r2.len) | t1∈r1.tup, t2∈r2
    .tup with t1[n1] = t2[n2]}⟩;

proc join3(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute):Relation
requires join_prec(r1, r2, n1, n2);
ensures join_spec(result,r1,r2,n1,n2); {
  var q:Relation := ⟨len: r1.len+r2.len, tup: choose s:Set[Row] with |s|=0⟩
      ;

  for t1 ∈ r1.tup do {
    for t2 ∈ r2.tup do {
      if t1[n1] = t2[n2] then {
        q.tup := q.tup ∪ {concat(t1, t2, r1.len, r2.len)};
      }
    }
  }

  return q;
}

fun join(r1:Relation, r2:Relation, n1:Attribute, n2:Attribute):Relation
requires join_prec(r1, r2, n1, n2);
ensures join_spec(result,r1,r2,n1,n2);
= join3(r1, r2, n1, n2);

/////////////////////////////////////////////

theorem join_cartesian_subset(r1:Relation, r2:Relation, n1:Attribute, n2:
    Attribute)
requires n1<r1.len ∧ n2<r2.len ∧ r1.len+r2.len ≤ N ∧ |r1.tup|*|r2.tup| ≤ M;
    ⇔
join3(r1, r2, n1, n2).tup ⊆ cartesian3(r1, r2).tup;

/////////////////////////////////////////////

rectype(L) List = nil | node(Attribute, List);
rectype(D) Query = from(TableId)
          | on(Attribute, Attribute, Query, Query)
          | where(Attribute, Element, Query)
          | select(List, Query)
          | add(Query, Query)
          | inters(Query, Query)
          | minus(Query, Query)
          | cart(Query, Query);


proc attributes(att:List):Array[N,Length] {
  var arr:Array[N,Length] := Array[N,Length](N);

  var l:List := att;
  var i:Attribute := 0;

  while l ≠ List!nil do {
    arr[i] := match l with {
      node(a:Attribute, li:List) -> a;
    };
    l := match l with {
      node(a:Attribute, li:List) -> li;
    };
    i := i+1;
  }

  return arr;
}

proc query(db:Database, q:Query):Relation {
  var r:Relation := match q with {
    from(tid:TableId) -> db[tid];
    on(n1:Attribute, n2:Attribute, q1:Query, q2:Query) -> join(query(db, q1
       ), query(db, q2), n1, n2);
    where(a:Attribute, e:Element, q:Query) -> select(query(db, q), a, e);
    select(a:List, q:Query) -> project(query(db, q), attributes(a));
    add(q1:Query, q2:Query) -> rUnion(query(db, q1), query(db, q2));
    inters(q1:Query, q2:Query) -> rIntersect(query(db, q1), query(db, q2));
    minus(q1:Query, q2:Query) -> rMinus(query(db, q1), query(db, q2));
    cart(q1:Query, q2:Query) -> cartesian(query(db, q1), query(db, q2));
  };
  return r;
}

/////////////////////////////////////////////

proc test():Relation {
  var dum:Map[Attribute,Element] := Map[Attribute,Element](0);

  var emp:Relation := ⟨len: 0, tup: choose s:Set[Row] with |s|=0⟩;
  var r1:Relation := ⟨len: 3, tup: choose s:Set[Row] with |s|=0⟩;
  var r2:Relation := ⟨len: 2, tup: choose s:Set[Row] with |s|=0⟩;

  r1.tup := r1.tup ∪ {dum};
  r2.tup := r2.tup ∪ {dum};
  dum[1] := 1;
  r1.tup := r1.tup ∪ {dum};
  r2.tup := r2.tup ∪ {dum};
  dum[0] := 1;
  r1.tup := r1.tup ∪ {dum};
  dum[1] := 0;
  r2.tup := r2.tup ∪ {dum};
  dum[1] := 1;
  dum[2] := 1;
  r1.tup := r1.tup ∪ {dum};

  var columns:List := List!node(0, List!node(2, List!nil));

  var db:Array[K, Relation] := Array[K, Relation](emp) with [0]=r1 with
      [1]=r2;

  return query(db, Query!on(0,0,
      Query!select(columns,
       Query!where(1,1,
        Query!from(0)
       )
      ),
        Query!from(1)
       )
      );
}

proc result():Relation {
  var dum:Map[Attribute,Element] := Map[Attribute,Element](0);
  var r:Relation := ⟨len: 4, tup: choose s:Set[Row] with |s|=0⟩;

  r.tup := r.tup ∪ {dum};

  dum[3] := 1;
  r.tup := r.tup ∪ {dum};

  dum[3] := 0;
  dum[0] := 1;
  dum[2] := 1;
  r.tup := r.tup ∪ {dum};

  dum[1] := 1;
  r.tup := r.tup ∪ {dum};

  return r;
}

theorem correct_result() ⇔ test() = result();

proc query_to_sql(q0:Query):()
proc translate_on(n1:Attribute, n2:Attribute, q0_1:Query, q0_2:Query):()
proc translate_cart(q0_1:Query, q0_2:Query):()
proc translate_where(a:Attribute, e:Element, q0:Query):()
proc translate_from(tid:TableId):()
proc translate_blank(q0:Query):()
proc translate_select(l:List, q0:Query):()

proc translate_add(q0_1:Query, q0_2:Query):() {
  print "(";
  query_to_sql(q0_1);
  print ") UNION";
  print "(";
  query_to_sql(q0_2);
  print ")";
}

proc translate_inters(q0_1:Query, q0_2:Query):() {
  print "(";
  query_to_sql(q0_1);
  print ") INTERSECT";
  print "(";
  query_to_sql(q0_2);
  print ")";
}

proc translate_select(l:List, q0:Query):() {
  var att:List := l;
  print "SELECT";
  while att ≠ List!nil do {
    var a:Attribute := match att with {
      node(a:Attribute, li:List) -> a;
    };
    att := match att with {
      node(a:Attribute, li:List) -> li;
    };
    if att ≠ List!nil then {
      print "{1}, ", a;
    } else {
      print a;
    }
  }
  match q0 with {
    where(a:Attribute, e:Element, q1:Query) -> {
      translate_where(a, e, q1);
    }
    on(n1:Attribute, n2:Attribute, q1_1:Query, q1_2:Query) -> {
      translate_on(n1, n2, q1_1, q1_2);
    }
    from(tid:TableId) -> {
      translate_from(tid);
    }
    _ -> {
      translate_blank(q0);
    }
  }
}

proc translate_blank(q0:Query):() {
  print "FROM (";
  query_to_sql(q0);
  print ")";
}

proc translate_from(tid:TableId):() {
  print "FROM {1}", tid;
}

proc translate_where(a:Attribute, e:Element, q0:Query):() {
  match q0 with {
    on(n1:Attribute, n2:Attribute, q1_1:Query, q1_2:Query) -> {
      translate_on(n1, n2, q1_1, q1_2);
    }
    from(tid:TableId) -> {
      translate_from(tid);
    }
    _ -> {
      translate_blank(q0);
    }
  }
  print "WHERE {1} = {2}", a, e;
}

proc translate_cart(q0_1:Query, q0_2:Query):() {
  var q1_term:Tuple[Bool, TableId] := match q0_1 with {
    from(tid:TableId) -> ⟨true, tid⟩;
    _ -> ⟨false, 0⟩;
  };
  var q2_term:Tuple[Bool, TableId] := match q0_2 with {
    from(tid:TableId) -> ⟨true, tid⟩;
    _ -> ⟨false, 0⟩;
  };
  print "SELECT *";
  if q1_term.1 then {
    if q2_term.1 then {
      print "FROM {1}, {2}", q1_term.2, q2_term.2;
    } else {
      print "FROM {1}, (", q1_term.2;
      query_to_sql(q0_2);
      print ")";
    }
  } else {
    if q2_term.1 then {
      print "FROM (";
      query_to_sql(q0_1);
      print "), {1}", q2_term.2;
    } else {
      print "FROM (";
      query_to_sql(q0_1);
      print "), (";
      query_to_sql(q0_2);
      print ")";
    }
  }
}


proc translate_on(n1:Attribute, n2:Attribute, q0_1:Query, q0_2:Query):() {
  var q1_term:Tuple[Bool, TableId] := match q0_1 with {
    from(tid:TableId) -> ⟨true, tid⟩;
    _ -> ⟨false, 0⟩;
  };
  var q2_term:Tuple[Bool, TableId] := match q0_2 with {
    from(tid:TableId) -> ⟨true, tid⟩;
    _ -> ⟨false, 0⟩;
  };

  if q1_term.1 then {
    if q2_term.1 then {
      print "FROM {1}", q1_term.2;
      print "JOIN {1}", q2_term.2;
    } else {
      print "FROM {1}", q1_term.2;
      print "JOIN (";
      query_to_sql(q0_2);
      print ")";
    }
  } else {
    if q2_term.1 then {
      print "FROM (";
      query_to_sql(q0_1);
      print ")";
      print "JOIN {1}", q2_term.2;
    } else {
      print "FROM (";
      query_to_sql(q0_1);
      print ")";
      print "JOIN (";
      query_to_sql(q0_2);
      print ")";
    }
  }
  print "ON {1} = {2}", n1, n2;
}

proc query_to_sql(q0:Query):() {
  match q0 with {
    add(q1_1:Query, q1_2:Query) -> {
      translate_add(q1_1, q1_2);
    }
    inters(q1_1:Query, q1_2:Query) -> {
      translate_inters(q1_1, q1_2);
    }
    minus(q1_1:Query, q1_2:Query) -> {
      print "Not implemented";
    }
    cart(q1_1:Query, q1_2:Query) -> {
      translate_cart(q1_1, q1_2);
    }
    select(l:List, q1:Query) -> {
      translate_select(l, q1);
    }
    where(a:Attribute, e:Element, q1:Query) -> {
      print "SELECT *";
      translate_where(a, e, q1);
    }
    on(n1:Attribute, n2:Attribute, q1_1:Query, q1_2:Query) -> {
      print "SELECT *";
      translate_on(n1, n2, q1_1, q1_2);
    }
    from(tid:TableId) -> {
      print "SELECT *";
      translate_from(tid);
    }
  }
}

proc sample_database():Array[K, Relation] {
  var dum:Map[Attribute,Element] := Map[Attribute,Element](0);

  var emp:Relation := ⟨len: 0, tup: choose s:Set[Row] with |s|=0⟩;
  var r1:Relation := ⟨len: 2, tup: choose s:Set[Row] with |s|=0⟩;
  var r2:Relation := ⟨len: 2, tup: choose s:Set[Row] with |s|=0⟩;

  r1.tup := r1.tup ∪ {dum};
  dum[1] := 1;
  r1.tup := r1.tup ∪ {dum};
  r2.tup := r2.tup ∪ {dum};
  dum[0] := 1;
  r2.tup := r2.tup ∪ {dum};
  dum[1] := 0;
  r2.tup := r2.tup ∪ {dum};

  var db:Array[K, Relation] := Array[K, Relation](emp) with [0]=r1 with
      [1]=r2;

  return db;

}

proc query_to_sql_atomic_test():() {

```

```
var db:Array[K, Relation] := sample_database();

var q1:Query := Query!select(List!node(2, List!node(0, List!nil)), Query!
    where(1, 0, Query!on(0, 1, Query!from(0), Query!from(1))));
var q2:Query := Query!select(List!node(1, List!node(0, List!nil)), Query!
    where(1, 1, Query!from(0)));
var q3:Query := Query!select(List!node(1, List!node(2, List!nil)), Query!
    on(0, 1, Query!from(0), Query!from(1)));
var q4:Query := Query!select(List!node(1, List!node(0, List!nil)), Query!
    from(1));
var q5:Query := Query!where(1, 0, Query!on(0, 1, Query!from(0), Query!
    from(1)));
var q6:Query := Query!where(1, 0, Query!from(0));
var q7:Query := Query!on(0, 1, Query!from(0), Query!from(1));
var q8:Query := Query!from(0);

print "Query result:";
print query(db, q1);
query_to_sql(q1);
print ";";
print "";

print "Query result:";
print query(db, q2);
query_to_sql(q2);
print ";";
print "";

print "Query result:";
print query(db, q3);
query_to_sql(q3);
print ";";
print "";

print "Query result:";
print query(db, q4);
query_to_sql(q4);
print ";";
print "";

print "Query result:";
print query(db, q5);
```

```
  query_to_sql(q5);
  print ";";
  print "";

  print "Query result:";
  print query(db, q6);
  query_to_sql(q6);
  print ";";
  print "";

  print "Query result:";
  print query(db, q7);
  query_to_sql(q7);
  print ";";
  print "";

  print "Query result:";
  print query(db, q8);
  query_to_sql(q8);
  print ";";
  print "";

}

proc query_to_sql_nonatomic_test():() {

  var db:Array[K, Relation] := sample_database();

  var qq1:Query := Query!select(List!node(0, List!nil), Query!select(List!
      node(1, List!node(0, List!nil)), Query!from(1)));
  var qq2:Query := Query!where(1, 0, Query!select(List!node(1, List!node(0,
       List!nil)), Query!from(1)));
  var qq3:Query := Query!on(1, 0, Query!select(List!node(1, List!node(0,
      List!nil)), Query!from(1)), Query!select(List!node(0, List!nil),
      Query!from(0)));
  var qq4:Query := Query!on(1, 0, Query!where(1, 0, Query!from(0)), Query!
      select(List!node(0, List!nil), Query!from(0)));
  var qq5:Query := Query!on(0, 1, Query!where(1, 0, Query!from(0)), Query!
      where(0, 0, Query!from(0)));
  var qq6:Query := Query!where(0, 0, Query!where(1, 1, Query!from(0)));

  print "Query result:";
  print query(db, qq1);
  query_to_sql(qq1);
  print ";";
  print "";

  print "Query result:";
  print query(db, qq2);
  query_to_sql(qq2);
  print ";";
  print "";

  print "Query result:";
  print query(db, qq3);
  query_to_sql(qq3);
  print ";";
  print "";

  print "Query result:";
  print query(db, qq4);
  query_to_sql(qq4);
  print ";";
  print "";

  print "Query result:";
  print query(db, qq5);
  query_to_sql(qq5);
  print ";";
  print "";

  print "Query result:";
  print query(db, qq6);
  query_to_sql(qq6);
  print ";";
  print "";

}
```

Listing A.1: Complete RISCAL source code

B. Test Architecture

- **Host system**
  - **OS:** Windows 10 Version 22H2
  - **CPU:** Intel(R) Core(TM) i3-7100 CPU @ 3.90GHz (2 cores)
  - **RAM:** 16000 MB
- **Virtual Machine**
  - **OS:** Debian GNU/Linux 11 (bullseye)
  - **RAM:** 11370 MB
- **Java:** openjdk 11.0.18 2023-01-17
- **RISCAL:** Version 4.2.2 (September 16, 2022)

Bibliography

[1] Gábor Bodnár. *Information Systems Lecture Notes*. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria. 2005. URL: `https://www3.risc.jku.at/education/courses/ws2011/is/ln.pdf`.

[2] Andreas Meier and Michael Kaufmann. *SQL- & NoSQL-Datenbanken*. 8., überarb. u. erw. Aufl. 2016. eXamen.press. Berlin, Heidelberg: Springer Vieweg, 2016. ISBN: 978-3-662-47664-2. DOI: `10.1007/978-3-662-47664-2`.

[3] E. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13 (Jan. 1970), pp. 377–387. DOI: `10.1007/978-3-642-48354-7_4`.

[4] E. Codd. "Relational completeness of database sublanguages". In: *ACM Transactions on Database Systems - TODS* (Jan. 1971), pp. 6–14. URL: `http://www.inf.unibz.it/~franconi/teaching/2006/kbdb/Codd72a.pdf`.

[5] E. Codd. "Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks". In: *SIGMOD Record* 38 (June 2009), pp. 17–36. DOI: `10.1145/1558334.1558336`.

[6] *Select-Project-Join Expressions*. URL: `https://mlwiki.org/index.php/Select-Project-Join_Expressions`.

[7] Wolfgang Schreiner. *Thinking Programs: Logical Modeling and Reasoning about Languages, Data, Computations, and Executions*. Springer, 2021. DOI: `10.1007/978-3-030-80507-4`.

[8] Wolfgang Schreiner. *Concrete Abstractions: Formalizing and Analyzing Discrete Theories and Algorithms with the RISCAL Model Checker*. Springer, 2023. ISBN: 978-3-031-24933-4. DOI: `10.1007/978-3-031-24934-1`.

[9] *RISCAL*. 2022. URL: `https://www3.risc.jku.at/research/formal/software/RISCAL/`.

[10] Wolfgang Schreiner (with contributions of Franz-Xaver Reichl and Ágoston Sütő). *The RISC Algorithm Language (RISCAL), Tutorial and Reference Manual*. Tech. rep. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, Mar. 2023. URL: `https://www3.risc.jku.at/research/formal/software/RISCAL/manual/main.pdf`.

[11] Clark Barret, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Department of Computer Science, The University of Iowa, 2017. URL: `https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf`.

[12] Franz-Xaver Reichl. "The Integration of SMT Solvers into the RISCAL Model Checker". MA thesis. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2020. URL: `https://epub.jku.at/obvulihs/content/titleinfo/5118206`.

[13] *RISCTP*. 2022. URL: `https://www3.risc.jku.at/research/formal/software/RISCTP/`.

[14] Wolfgang Schreiner. *The RISCTP Theorem Proving Interface Tutorial and Reference Manual*. Tech. rep. 22-07. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2023. DOI: `10.35011/risc.22-07`.

[15] Alfons Kemper and André Eickler. *Datenbanksysteme: Eine Einführung*. 10th ed. De Gruyter Studium. De Gruyter Oldenbourg, 2015. ISBN: 978-3-11-044375-2.

[16] M.J. Hernandez. *Database Design for Mere Mortals: A Hands-on Guide to Relational Database Design*. For Mere Mortals Series. Addison-Wesley, 2003. ISBN: 9780201752847.

[17] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, et al. "A History and Evaluation of System R". In: *Commun. ACM* 24.10 (Oct. 1981), pp. 632–646. ISSN: 0001-0782. DOI: `10.1145/358769.358784`.

[18] *SQLite*. URL: `https://sqlite.org/index.html`.

[19] Wolfgang Schreiner and Franz-Xaver Reichl. "Mathematical Model Checking Based on Semantics and SMT". In: *Transactions on Internet Research* 16(2) (2021), pp. 4–13. URL: `http://ipsitransactions.org/journals/papers/tir/2020jul/p2.pdf`.

[20] *Chinook Database*. URL: `https://github.com/lerocha/chinook-database`.

[21] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[22] Armin Biere, Marijn Heule, H. Maaren, et al. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Jan. 2009. ISBN: 1586039296, 9781586039295.

[23] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, et al. *Rigorous Software Development - An Introduction to Program Verification.* Undergraduate Topics in Computer Science. Springer, 2011. ISBN: 978-0-85729-018-2. DOI: `10.1007/978-0-85729-018-2`.

[24] Dirk Hoffmann. *Grenzen der Mathematik*. Jan. 2018. ISBN: 978-3-662-56616-9. DOI: `10.1007/978-3-662-56617-6`.

[25] E.M. Clarke, T.A. Henzinger, H. Veith, et al. *Handbook of Model Checking*. Springer, May 2018. DOI: `10.1007/978-3-319-10575-8`.

[26] Armin Biere, Alessandro Cimatti, Edmund Clarke, et al. "Bounded Model Checking". In: *Advances in Computers* 58 (Dec. 2003), pp. 117–148. DOI: `10.1016/S0065-2458(03)58003-2`.

[27] *SMT-LIB The Satisfiability Modulo Theories Library*. URL: `https://smtlib.cs.uiowa.edu/`.

[28] Wolfgang Schreiner. "Validating Mathematical Theorems and Algorithms with RISCAL". In: *Intelligent Computer Mathematics*. Ed. by Florian Rabe, William M. Farmer, Grant O. Passmore, et al. Cham: Springer International Publishing, 2018, pp. 248–254. ISBN: 978-3-319-96812-4. DOI: `10.1007/978-3-319-96812-4_21`.

[29] Wolfgang Schreiner and Franz-Xaver Reichl. "Semantic evaluation versus SMT solving in the RISCAL model checker". In: *RISC Report Series, 21-11*. June 2021. DOI: `10.35011/risc.21-11`.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, May 4, 2023 Joachim Borya