Analysis and Formalization of Typing in SQL Engines

Wenjia Ye
The University of Hong Kong
Hong Kong, China
yewenjia@connect.hku.hk

Matías Toro
Computer Science Department,
University of Chile
Santiago, Chile
mtoro@dcc.uchile.cl

Claudio Gutierrez
Computer Science Department,
University of Chile & IMFD
Santiago, Chile
cgutierr@dcc.uchile.cl

Bruno C. d. S. Oliveira
The University of Hong Kong
Hong Kong, China
bruno@cs.hku.hk

Éric Tanter
Computer Science Department,
University of Chile & IMFD
Santiago, Chile
etanter@dcc.uchile.cl

ABSTRACT

Practical SQL engines differ in subtle ways in their handling of typing constraints and implicit type casts. These issues, usually not considered in formal accounts of SQL, directly affect portability between engines. We present a general formal semantics framework for SQL, named RAFT, that explicitly captures the semantics of types, both static and dynamic. RAFT is expressed in terms of abstract operators that provide the necessary leeway to precisely model different SQL engines. We show that this formalism has good theoretical properties (e.g. practical conditions that ensure type safety and soundness) and is of practical use. Indeed, we provide RAFT instantiations for 5 major SQL engines: PostgreSQL, Microsoft SQL Server, MySQL, SQLite, and Oracle Database. To validate the adequacy of the formalism, we implement RAFT in Python, along with these five instances, and test them with thousands of randomly-generated queries. RAFT can serve as precise documentation of typing in existing engines and potentially guide their evolution, as well as provide a formal basis to study type-aware query optimizations, and design provably-correct query translators.

PVLDB Reference Format:
Wenjia Ye, Matías Toro, Claudio Gutierrez, Bruno C. d. S. Oliveira, and Éric Tanter. Analysis and Formalization of Typing in SQL Engines. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

Query translation between different SQL engines is a common practice arising in different scenarios, like database migration (to reduce costs, maintenance, changes in software, etc.) [7, 21, 23] and prototyping (code in lightweight databases like SQLlite and then port to a more robust database). Today there are many tools addressing this task [1, 2, 19].

Translation between SQL engines could bring many surprises. One of the most notorious is the semantic discrepancy between SQL engines related to typing behaviour, the problem we address in this paper. This problem is mainly due to differences in datatypes, type checking, when to perform the checks, and explicit and implicit type casts that may or may not be performed by the engines. This poses substantial challenges for developers and database administrators. Existing tools, whether paid or open source, often fall short in addressing these differences, tending to prioritize syntax over behavioral disparities.[1]

Table 1 presents a sample of minimal examples of this wide difference in behaviour. As these examples show, database engines have different treatment of types, that obey to different design models, like some being statically typed while others embrace dynamic typing, to different approaches to the overloading of basic arithmetic and comparison operations, etc. For now we only want to stress that there is a relevant problem here. In Section 2 we will give more detailed explanations for these cases.

Addressing and understanding these anomalies is not a simple task. On one hand, SQL standards do not cover many issues related to typing or leave the interpretation rather open.[2] On the other hand, many of the design decision of the engines are hidden under optimization mechanisms that either are not public or complex to find in the code. Last, but not least, the problem has not been addressed by the research literature. Indeed, although there is solid work on the formalization of the semantics of SQL [20, 31], they assume that all comparisons and operations apply to the right types. On the other hand, typing in SQL and syntax errors have been explored (particularly in the Programming Language community from where we took some inspiration [5, 14, 25, 27, 32], but the problem addressed of type constraints and casts which may induce errors at runtime, the one addressed in this paper, has not been

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[1]Some examples: https://en.wikibooks.org/wiki/Converting_MySQL_to_PostgreSQL http://www.sqlines.com/online https://www.rebasedata.com/convert-bmysql-bto-bpostgres-bonline

[2]A good example is the following: To cast an exact number to an exact numeric type, e.g. from a real to int, it specifies: "If there is a representation of SV [source value] in the data type TD [target value] that does not lose any leading significant digits after rounding or truncating if necessary, then TV is that representation. The choice of whether to round or truncate is implementation-defined." ANSI SQL 1992, Sec. 6.10, Case 3)a)i))

Table R

A	B
`'Bob'`	10
`'1'`	20
`'1.1'`	30

	Query	PSQL	MSSQL	Oracle	MySQL	SQLite
E1	`SELECT 1.1 + 1 FROM R`	2.1	2.1	2.1	2.1	2.1
E2	`SELECT '1' + 1 FROM R`	2	2	2	2	2
E3	`SELECT '1.1' + 1 FROM R`	✗	$\not\rightarrow$	2.1	2.1	2.1
E4	`SELECT '1.1' + 1.1 FROM R`	2.1	2.1	2.1	2.1	2.1
E5	`SELECT '1' + '1' FROM R`	✗	'11'	2	2	2
E6	`SELECT 'sql' + '2ra' FROM R`	✗	'sql2ra'	$\not\rightarrow$	2	2
E7	`SELECT 1+A FROM R WHERE B=20`	✗	3	3	3	3
E8	`SELECT 1+A FROM R WHERE B=10`	✗	$\not\rightarrow$	$\not\rightarrow$	1	1
E9	`SELECT 1 + A FROM (SELECT '2' AS A) B`	✗	3	3	3	3
E10	`SELECT 1 FROM R WHERE '1' < 2`	1	1	1	1	∅
E11	`SELECT 1 FROM R WHERE '1.1' < 2`	✗	$\not\rightarrow$	1	1	∅
E12	`SELECT '1.1' FROM R INTERSECT SELECT 1.1 FROM R`	1.1	1.1	✗	1.1	∅
E13	`SELECT '1.1' FROM R INTERSECT SELECT 1 FROM R`	✗	$\not\rightarrow$	✗	∅	∅

Table 1: Examples of discrepant behaviors of different database engines. The queries are artificially chosen to be minimal examples of typing issues. For simplicity we show only one element of the result. We use ✗ to denote a static error (before execution), and $\not\rightarrow$ to denote a runtime error.

dealt with. In Section 8 of related work we present with more detail these antecedents.

In this paper we address the problem by proposing a general formal framework (called RAFT) that models the intricate behavioral discrepancies across database engines. Our framework (1) formalizes the static semantics of queries via a type system, (2) makes implicit type casts explicit via an elaboration phase that introduces explicit type casts, and (3) formalizes the dynamic semantics of queries via a monadic evaluator. The type system, elaboration, and evaluator are expressed in terms of abstract operators. This framework provides, in theory and in practice, the necessary leeway to precisely model different SQL engines. To the best of our knowledge, no comprehensive modeling of these features exists.

From a practical point of view, we tested the model over five different and well-known SQL engines: PSQL, MSSQL, Oracle, MySQL and SQLite. It shows the flexibility and comprehensiveness of the framework, as it permits to understand different behaviors and thus opens the door for informed decisions when translating queries. The framework is tested with randomly generated queries. At the same time, the framework is a basis to formally discuss future typing-related design decisions for SQL, as it distills the core parameters involved.

From a theory point of view, we prove several formal results. First, a type safety result that proves that well typed queries either reduce to a table or raise a (controlled) type error. In other words, evaluation of well-typed queries does not get stuck. We enhance this result proving that the resulting table can be typed to the same type of the query. Moreover, PSQL, MySQL and SQLite satisfy a theorem that states that if the programmer do not uses explicit casts in well- typed queries, then the queries will evaluate without errors. Finally, regarding cast insertion, we can prove that independently of the engine (the proofs are parameterized by light constraints on abstract operators), translated queries preserve types, and the translation is unique.

In summary, this paper presents a formal and practical foundation for addressing the typing problems and challenges posed by database migration, by elucidating the database engine behavior in the presence of types, bridging the gap between various systems, and offering practical insights for both researchers and practitioners in the realm of database management.

The contributions and organization of the paper are as follows:

1. **Identification of Practical Issues:** In Section 2 we detect, localize and study the problems of semantic discrepancies due to typing, specifically type casts, which would affect query translation between SQL engines.
2. **Formal Semantics Framework:** We describe a formal framework, RAFT, based on a typed core subset of SQL with support for type casts. Section 3 overviews the overall design of RAFT and Section 4 provides the formal semantics of RAFT, which are expressed in terms of abstract operators that provide the necessary leeway to precisely model different SQL engines. We illustrate two different instantiations in Section 5, and provide three others in the supplementary material.
3. **Metatheory:** in Section 6 we present the metatheory of RAFT, demonstrating several important properties, such as the assurance that well-typed queries do not get stuck. For each property, we provide constraints on abstract operators required for these properties to hold. Furthermore, we illustrate which engine satisfy (or not) those properties.
4. **Empirical Validation:** In Section 7 we empirically validate our framework by developing multiple database interpreters. We test these interpreters with thousands of randomly-generated queries and compare their results with those obtained from real database engines. Additionally, we shed light on the impact and challenges of query optimizations on the evaluation process.

Section 8 discusses related work and Section 9 concludes. Additional material (proofs, full definitions, and prototype implementation) can be found in the supplementary material.

2 TYPING SEMANTIC DISCREPANCIES

In this work we study two kinds of different behavior between engines: (1) different results due to type conversions, and (2) different behavior related to type errors. Regarding the latter, we identify two kinds of type errors: *static errors*, which happen during compilation/before running the query; and *runtime errors* that occur during the execution of the query. We say that two engines differ in behavior if, for a given query, the results are different (including different kinds of errors).

To illustrate some of the differences, we explore different queries using a core subset of standard SQL. Like several other authors[9, 20, 28], this core comprises booleans, numbers, selections, cross-products, nested queries within `FROM` clauses, and set operations.

We also add support for arithmetic operators and type casts. Notably, features like nulls, aggregation, `EXISTS`, `IN`, and other forms of nested queries are not supported within this core. However, the aforementioned core serves to illustrate the primary distinctions, leaving the extension to these additional features as potential future work. For each query we illustrate their behavior in a mix of statically and dynamically typed engines such as PostgreSQL (PSQL), Microsoft SQL Server (MSSQL), MySQL, SQLite, and Oracle Database (Oracle).

The summary of examples and results can be found in Table 1. Relation `R` can be denoted using bags as `R` = {(`'Bob'`, `10`), (`'1'`, `20`), (`'1.1'`, `30`)}.

2.1 Arithmetic operations

For the first set of examples we focus on the expression being selected rather than the tables or the conditions.

E1 and E2. To begin, we present two examples that behave uniformly across the engines. For simplicity, we say that the result is 2.1, to denote a bag of uniform elements {2.1, 2.1, 2.1}.

E3. This is the first example that illustrates a difference in behavior. PSQL and MSSQL throw a type error, whereas the other engines return 2.1. The reason for the error is that in PSQL and MSSQL we can implicitly cast a string to an integer if the string is an integer, but not if the string is a real number. To fix this problem in PSQL, we must explicitly *cast* the string to a real number: `SELECT CAST('1.1' as FLOAT) + 1 FROM R`.

E4. If we take example 3, and change `1` for a real number, such as `1.1`, then the result is now 2.1 for every engine. The difference with respect to the previous example is that the plus operation is defined for both integers and real numbers, and now `'1.1'` can be cast directly to a real number.

E5. PSQL reports an static error due to the lack of an addition operator between two strings. MSSQL returns `'11'` as addition is overloaded for string. The other engines return 2 as addition is only defined between numbers, thus implicit casting `'1'` to `1`.

E6. Both PSQL and Oracle report an error, MSSQL uses string concatenation yielding `'sql2ra'`, and MySQL and SQLite yield `2`. The reason for the latter is due to the way these engines cast strings to numbers: they search for a number in the prefix of the string (if nothing is found then 0 is returned).

E7. PSQL rejects this query as column `A` is of type String, and the addition between numbers and strings is not defined. This behavior is more conservative than E2, as now it cannot determine statically if the given string can be cast to number or not. Other engines defer the check to runtime and return `2`. To fix this query in PSQL we can explicitly cast column `A` to integer: `SELECT 1+CAST(A as INT) FROM R WHERE B=20`.

E8. PSQL (statically) rejects this query similarly to E7. MSSQL and Oracle now fail dynamically as they cannot convert `'Bob'` to a number. MySQL and SQLite on the other hand do not fail and return `1` as `'Bob'` is cast to `0`. Casting `A` to `INT` in PSQL would make the error dynamic, and a runtime type error would be reported instead, similarly to MYSQL and SQLITE.

E9. Contrary to E2, PSQL raises a static error as the nested query hides the actual String returned by the subquery. All other engines yield `3` as conversions are optimistically performed at runtime. To fix this query in PSQL an explicit cast must be inserted `SELECT 1 + CAST(A AS INT) FROM (SELECT '2' AS A) B`.

Operation	SQLite	**Other engines**
`0 < 1`	`1`	`t`
`'0' < 1`	`0`	`t`
`'1' < 0`	`0`	`f`
`'0'+0 < 1`	`1`	`t`
`'0' < CAST(1 AS INT)`	`1`	`t`
`'0' < 1 + 0`	`0`	`t`

Table 2: Behavior of the comparison operator in SQLite

2.2 Boolean operations

The following examples illustrate difference in behavior related to comparison operators on conditionals.

E10. Almost every engine is able to cast `'1'` to `INT`, returning 1. SQLite, on the other hand, returns a empty result as the condition is `false`. This is because SQLite does not perform implicit conversions at the boundaries of comparisons. SQLite has a type hierarchy where every string is bigger than any number.

E11. Now, if the left operand is a string representing a real, then PSQL and MSSQL return a type error. This is because the best type of the comparison operator is the one that takes two integers as argument (because 2 is an integer). As there is no direct implicit conversion between a string representing a real and an integer the query is rejected. SQLite still returns an empty result, and MySQL and Oracle returns 1.

Comparison operator in SQLite. SQLite warrants special attention to illustrate unique cases related to the comparison operator, as exemplified in Table 2. Notably, operations `'0' < 1` and `'1' < 1` yields `0`. This behavior is attributed to the fact that strings are considered larger than integers, as previously explained. However, When an arithmetic operation or a cast is introduced the expressions now yields `1`. One might then expect that `'0' < 1+0` would yield `1`. Nevertheless, the result is actually `0`, since the result of the arithmetic operation remains uncertain.

2.3 Set operations

E12. PSQL, MSSQL and MySQL, yield `1.1`. This is because, during intersection, two conditions are checked: (1) the cardinality of both subqueries must match, and (2) the types of the columns must also be consistent. To achieve the second condition, an implicit conversion from string to real is inserted in the left subquery (the other direction is forbidden). However, Oracle encounters a type error since the column types do not match. On the other hand, SQLite returns an empty result as `1.1` is not the same as `'1.1'`.

E13. Now as `'1.1'` cannot be implicitly cast to an integer (the column type of the right subquery), this program is rejected by PSQL, MSSQL and Oracle. Both MySQL and SQLite return an empty result.

Having explored some discrepancies in semantics across different database engines due to types, we now transition to presenting an overview of our solution.

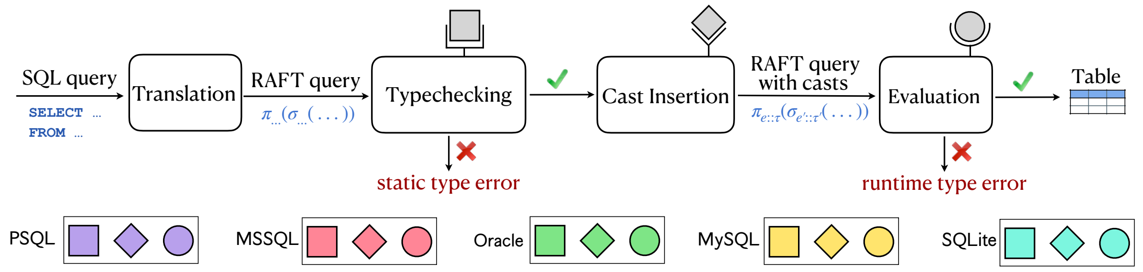

Figure 1: Overview of RAFT

3 OVERVIEW OF RAFT

In this paper, we introduce RAFT, a general formal semantics framework for SQL that explicitly captures the semantics of types, both static and dynamic. Figure 1 depicts a high-level overview of the framework. The processing of an SQL query involves four sequential phases: translation, typechecking, cast insertion, and evaluation.

Translation. In the translation step (Section 4.6), an SQL query is (1) analyzed to rule out syntactically invalid queries such as `SELECT FROM FROM` and (2) transformed to a RAFT query. RAFT is a core typed relational algebra (Section 4), with classical features such as projections, selections, and set operations, and unusual features such as arithmetic operations and casts.

Typechecking. RAFT does not assume the absence of type mismatches; users can write queries such as `SELECT R.A from R WHERE 'Bob' = 1`. For this reason, a typechecker validates queries before evaluation (Section 4.4). This typechecker is responsible for identifying mismatches between types and the number of columns of subqueries in set operations, and to ensure proper access to column names in scope. More importantly, it validates the appropriate usage of both implicit and explicit casts, rejecting operations that may result in casts known to fail during evaluation. If the typechecker rejects the query, the process terminates and reports a static type error to the user. However, if the query is deemed well-typed, it proceeds to the third phase.

Cast Insertion. The cast insertion phase (Section 4.5) transforms a RAFT query by making all implicit type casts explicit. The purpose of this phase is to circumvent the complexities that would arise from handling implicit casts in the evaluation phase, thereby simplifying the dynamic semantics of RAFT. This approach aligns the dynamic semantics closely with the classical relational algebra semantics, while only necessitating an extension to accommodate explicit casts and errors.

Evaluation. The evaluation phase (Section 4.3) interprets the transformed RAFT query. For this purpose, we present a monadic evaluator expressed using denotational semantics, cleanly encompassing the management of explicit type conversions and runtime errors. Should a type error arise during evaluation, the evaluation process is halted, and a runtime type error is reported. In the absence of runtime type errors, the outcome of evaluation is a table.

Parametrization. RAFT is meant to precisely capture the actual semantics of different SQL engines. To do so, typechecking, cast insertion, and evaluation are parametrized by engine-specific components (Figure 1). Specifically, these phases are expressed in terms of abstract operators that provide the necessary leeway to precisely model different SQL engines (Section 4.2), and represented by the different shapes in the figure. These operators include: how to cast a given value to a specific type, how to resolve overloading (*i.e.* the use of an operator with different argument types), determining the types of values and operations, and when to insert explicit casts. We provide instantiations of these operators for 5 major SQL engines: PSQL, MSSQL, Oracle, MySQL and SQLite. Section 5 presents RAFT/PSQL and RAFT/SQLite in full. The instantiations of RAFT for the other three engines are provided in the supplementary material.

Properties. Providing a formal framework makes it possible to precisely formulate and prove properties satisfied by all or some engines (Section 6). Some properties can be proven generally at the level of RAFT, assuming some basic conditions of the abstract operators; consequently, any instantiation that respects these conditions is guaranteed to enjoy the said properties. Others can be shown of specific instantiations. In particular, we show that every well-typed RAFT query does not get stuck (such as in queries which select columns that do not exist), either returning a table or raising a runtime type error.

We prove that—with the exception of SQLite—the table resulting from the evaluation of a well-typed query indeed has the predicted type. Additionally, we establish that the absence of explicit casts in the original source SQL query guarantees that no type errors can occur at runtime. We show that this requirements are satisfied by PSQL, MySQL, and SQLite, but not by MSSQL and Oracle.

Finally, we prove that cast insertion is a type-preserving function. This means that cast insertion is deterministic, and that the resulting query is typed to the same type of the source query. We prove that it holds for all the engines considered in this work.

Implementation. To validate the adequacy of the formalism, we implemented RAFT in Python along with these five engine instances. We randomly-generated thousands of random queries, corroborating that queries behave the same in RAFT as in the corresponding engines. We validate that, disregarding optimizations, the behavior of queries remains consistent: either both the prototype and the real engine encounter errors, or both evaluate to the same table. Additionally, considering optimizations, we validate that when both the prototype and the engine successfully evaluate a query, the results match.

Non-goals. Finally, it is important to note that several *non-goals* we set for the development of RAFT include performance and query optimization, and are left as future work. Our primary focus centers on exploring the semantics and properties of the framework as we discuss next.

		Types and Schemas	
τ	::=	$\mathbb{R} \mid \mathbb{Z} \mid \mathbb{B} \mid \mathsf{String} \mid ?$	(value types)
T	::=	$N \mapsto \tau \mid T, N \mapsto \tau$	(relation types)
Γ	::=	$\emptyset \mid \Gamma, R \mapsto T$	(schema)
		Values and Expressions	
$\mathsf{O_A}$	$\in$	$\{+\}$	(arithmetic ops)
$\mathsf{O_C}$	$\in$	$\{<, =\}$	(comparison ops)
$\mathsf{O_B}$	$\in$	$\{\wedge, \vee\}$	(boolean ops)
w	::=	$r \mid n \mid b \mid s$	(simple values)
v	::=	$w \mid w :: ?$	(values)
e	::=	$N \mid v \mid e\, \mathsf{O_A}\, e \mid e :: \tau$	(general expressions)
θ	::=	$e\, \mathsf{O_C}\, e \mid \theta\, \mathsf{O_B}\, \theta \mid \neg \mathsf{O_B}$	(boolean expressions)
β	::=	$e \text{ as } N \mid \beta, e \text{ as } N$	(aliased expressions)
		Queries	
$\mathsf{O_S}$	$\in$	$\{\cup, \cap, \times, -\}$	(set query ops)
Q	::=	$R \mid \pi_\beta(Q) \mid \sigma_\theta(Q) \mid Q\, \mathsf{O_S}\, Q$	(queries)

Figure 2: Syntax of RAFT. r, n, b, s **denote respectively a real number, an integer number, a Boolean value and a string.** N **is a name.** R **a relation.**

4 THE RAFT FORMAL FRAMEWORK

In this section we present the syntax, dynamic semantics, type system, cast insertion and translation of RAFT. A RAFT query is expressed using a core typed relational algebra, supporting projections, selection, set operations, arithmetic and boolean operations, and implicit and explicit casts.

4.1 Syntax

The syntax of RAFT is presented in Figure 2. The formalization is inspired by the work of Guagliardo and Libkin [20], except that here we deal with typing instead of assuming a prior (unstudied) typing phase. Types play a central role in this work, because as we illustrated, typing discrepancies are a source of behavioral differences between engines. This section first presents types and schemas, then values and expressions, and finally queries.

Types and Schemas. There are two category of types: *value types* τ that are used to type expressions (and values), and *relation types* T to type queries and tables (relations). A value type τ is either a real $\mathbb{R}$, an integer $\mathbb{Z}$, a boolean $\mathbb{B}$ or a string String. We use the symbol ? for the unknown type, used by PSQL to type string literals [18], and to model flexible typing in SQLite. A relation type T is an ordered list of pairs of the form $N : \tau$, where N is a column name and τ its corresponding value type. For instance, $T_1 = \mathsf{Name} \mapsto \mathsf{String}, \mathsf{Age} \mapsto \mathbb{Z}$ and $T_2 = \mathsf{Name} \mapsto \mathsf{String}, \mathsf{Price} \mapsto \mathbb{R}$ are relation types. A schema Γ is a list of pairs R, T where R is a relation name and T a relation type. Intuitively, schemas represent types of databases. For instance, the schema of a database of persons and products can be written as $\mathsf{Person} \mapsto T_1, \mathsf{Product} \mapsto T_2$.

Values and expressions. We represent tables as sets of *rows*, where each row is a list of *values*[3]. A *simple value* w, which represents an atomic piece of data, is either a real number r, a natural number n, a boolean b or a string s. Values v are either a simple value w, or a simple value cast to the unknown type $w :: ?$ (the latter is not used directly by programmers, and it is used exclusively in engines such as SQLite). Expressions, which can be reducible, are classified into three categories: *general expressions*, *boolean expressions*, and *aliased expressions.* A general expression e, is used in projections and selections, and is either a column name N such as Name or Age, a value v, an arithmetic operation $e\, \mathsf{O_A}\, e$ (for simplicity we only use $+$), or an explicit cast $e :: \tau$. Boolean expressions θ are used exclusively as conditions in selections, and are either comparison operations $e\, \mathsf{O_C}\, e$, or logical combinations of them. Note that we do not include boolean expressions as general expressions, because some engines do not support selecting boolean values in queries (e.g. in MSSQL and Oracle, `SELECT 1 < 2 FROM R` is a syntactically invalid query while it is valid in PSQL, MySQL and SQLite). An aliased expression β is a (non-empty) list of pairs e as N, where e is an expression and N its alias. Note that this is a richer version than the classical relational algebra aliasing operation, which allows binding of names to expressions, instead of only to queries. For example, in `SELECT A.C FROM (SELECT 1+1 AS C FROM R) A`, the expression 1+1 gets a name `C` that can be used in the outer query.

Queries. A query is either a relation R, a projection $\pi_\beta(Q)$, a selection $\sigma_\theta(Q)$, or a set operation. Set operations include cross product $(Q \times Q)$, intersection $(Q \cap Q)$, union $(Q \cup Q)$ and difference $(Q - Q)$. Note that projection is non-standard as it is parametrized by a list of aliased expressions, instead of a list of names. This allows RAFT to model SQL queries like `SELECT 1+1 AS C FROM R`, which projects the arithmetic expression $1 + 1$ from every row of R, as $\pi_{(1+1) \text{ as } C}(R)$.

4.2 Abstract operators

As mentioned in Section 3, certain key operators in RAFT are left abstract, as they depend on the specific engine being used.

Bidirectional Implicit Cast. Operator $biconv(e_1, \tau_1, e_2, \tau_2) = \tau_3$ determines the optimal implicit type cast for a set operator applied to two columns of (possibly) different types. The *biconv* operator takes as argument two expressions (e_1, e_2) and their corresponding types (τ_1, τ_2), returning a type. It either returns τ_2 or τ_1 by testing if e_1 can be implicitly cast to τ_2, or if e_2 can be implicitly cast to τ_1 respectively.

Explicit Cast. Operator $cast(v, \tau) = v'$ attempts to cast value v to a value v' of type τ. This function is primarily used to evaluate casts at runtime.

Overloading Resolution. Operator $resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4$ determines which specific operation among a set of overloaded ones should be called based on the provided arguments during invocation. More specifically, given a set of function types $\overline{\tau_3} = \overline{\tau_5 \times \tau_6 \to \tau_7}$, it tries to find the best candidate type for expressions e_1 and e_2, typed as τ_1 and τ_2 respectively. A function type of the form $\tau_5 \times \tau_6 \to \tau_7$, denotes a function that accepts two arguments, of types τ_5 and τ_6 respectively, and produces a result of type τ_7.

Explicit Cast Feasibility. This operator takes one expression and two types, and returns a boolean. For simplicity, it is presented as a relation $e : \tau' \rightsquigarrow \tau$, and rules out explicit casts that are known to fail at runtime. It test if it is possible to explicitly cast expression e of type τ', to an expression of type τ.

Type of Values. Operator $ty(v) = \tau$ computes the type of values. *Candidate Types.* Operation $ty(\mathsf{O}) = \overline{\tau}$ determines the set of possible

[3]For simplicity we represent tables as sets, but the supplementary material presents rules using multisets.

$$\begin{aligned}
[\![R]\!]_{\mathsf{D}} &= \mathbf{ok}\ \mathsf{D}\,(R) && \text{(R}R\text{)}\\
[\![Q_1\ \mathsf{O}_{\mathsf{S}}\ Q_2]\!]_{\mathsf{D}} &= \mathrm{do}\big\{t_1 \leftarrow [\![Q_1]\!]_{\mathsf{D}}; t_2 \leftarrow [\![Q_2]\!]_{\mathsf{D}}; \mathbf{ok}\ t_1\ \mathsf{O}_{\mathsf{S}}\ t_2\big\} && \text{(R}\mathsf{O}_{\mathsf{S}}\text{)}\\
[\![\pi_{\beta}(Q)]\!]_{\mathsf{D}} &= \mathrm{do}\big\{t \leftarrow [\![Q]\!]_{\mathsf{D}}; \lceil\{[\![\beta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} | \mathsf{r} \in t\}\rceil\big\} && \text{(R}\pi\text{)}\\
[\![\sigma_{\theta}(Q)]\!]_{\mathsf{D}} &= \mathrm{do}\big\{t \leftarrow [\![Q]\!]_{\mathsf{D}}; \_ \leftarrow \lceil\{[\![\theta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} | \mathsf{r} \in t\}\rceil; && \text{(R}\sigma\text{)}\\
&\qquad \mathbf{ok}\ \{\mathsf{r} | \mathsf{r} \in t \wedge [\![\theta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}}\}\big\}\\
[\![e\ \mathsf{as}\ N]\!]_{\eta} &= [\![e]\!]_{\eta} && \text{(Ras)}\\
[\![\beta, e\ \mathsf{as}\ N]\!]_{\eta} &= \mathrm{do}\big\{\mathsf{r} \leftarrow [\![\beta]\!]_{\eta}; v \leftarrow [\![e]\!]_{\eta}; \mathbf{ok}\ \mathsf{r}, v\big\} && \text{(R}\beta\text{)}\\
[\![N]\!]_{\eta} &= \mathbf{ok}\ \eta\,(N) && \text{(R}N\text{)}\\
[\![v]\!]_{\eta} &= \mathbf{ok}\ v && \text{(R}v\text{)}\\
[\![v :: \tau]\!]_{\eta} &= \left|\begin{array}{ll} \mathbf{ok}\ v' & \text{if } \mathit{cast}(v, \tau) = v' \\ \mathbf{error} & \text{otherwise} \end{array}\right. && \text{(R}v\ ::\text{)}\\
[\![e :: \tau]\!]_{\eta} &= \mathrm{do}\big\{v \leftarrow [\![e]\!]_{\eta} \quad e \neq v; [\![v :: \tau]\!]_{\eta}\big\} && \text{(R}e\ ::\text{)}\\
[\![\theta_1\ \mathsf{O}_{\mathsf{B}}\ \theta_2]\!]_{\eta} &= \mathrm{do}\big\{b_1 \leftarrow [\![\theta_1]\!]_{\eta}; b_2 \leftarrow [\![\theta_2]\!]_{\eta}; \mathbf{ok}\ b_1\ \mathsf{O}_{\mathsf{B}}\ b_2\big\} && \text{(R}\mathsf{O}_{\mathsf{B}}\text{)}\\
[\![\neg\theta]\!]_{\eta} &= \mathrm{do}\big\{v_1 \leftarrow [\![\theta_1]\!]_{\eta}; \mathbf{ok}\ \neg b_1\big\} && \text{(R}\neg\text{)}\\
[\![e_1\ \mathsf{O}\ e_2]\!]_{\eta} &= \mathrm{do}\big\{v_1 \leftarrow [\![e_1]\!]_{\eta}; v_2 \leftarrow [\![e_2]\!]_{\eta}; && \text{(R}\mathsf{O}\text{)}\\
&\qquad \mathbf{ok}\ \mathit{apply}(\mathsf{O}, v_1, v_2) \quad \mathsf{O} \in \mathsf{O}_{\mathsf{A}} \cup \mathsf{O}_{\mathsf{C}}\big\}
\end{aligned}$$

**Figure 3: Dynamic Semantics of RAFT.
(abstract operators highlighted in gray)**

function types available for operation O. The output is a list of function types $\bar{t}$ given that many arithmetic and comparison operators are overloaded with more than one type.

Unknown Type Removing. Operator $\mathit{clean}(T) = T'$ performs post-processing for relation type T returning a new relation type T'. In some engines, this operation replaces occurrences of the unknown type with another type.

Annotation Insertion. Operator $\mathit{insert}(e, \tau, \mathsf{O}) = e'$ attempts to explicitly cast expression e to type τ in a context within the O operation. It returns a new expression e' depending on whether the expression was cast or not.

Value Operation Application. Operator $\mathit{apply}(\mathsf{O}, v_1, v_2) = v_3$ performs arithmetic or comparison operation O to values v_1 and v_2, yielding value v_3.

In the next subsections, we use these operators to define the dynamic semantics, the type system and the cast insertion procedure. Examples of instantiation of these abstract operators can be found in Section 5.

4.3 Dynamic Semantics

We now present the dynamic semantics of RAFT, described in Figure 3. As standard in programming languages, the *dynamic* semantics specify the way programs are evaluated (a.k.a the runtime), while the *static* semantics refer to typing. Here, the dynamic semantics define the evaluation of expressions, operators and queries. Typing for RAFT is presented in Section 4.4.

We follow standard notations in database and programming languages [3, 20, 30]. The dynamic semantics of RAFT differ from the ones of Guagliardo and Libkin [20] (GL) as follows. First, RAFT dynamic semantics are defined over a relational algebra, whereas GL uses SQL syntax. Second, RAFT supports implicit and explicit casts, and arithmetic operations, but not null values, and `DISTINCT`, `IN`, and `EXISTS` operators supported by GL. Finally, and more importantly, as some casts may be invalid, the dynamic semantics must deal with the possibility of runtime errors. For instance, in PSQL the query `SELECT CAST('s' as INT) FROM R` ($\sigma_{\texttt{'s'}::\mathbb{Z}}(R)$) evaluates to an error.

To concisely account for the possibility of runtime errors, we present the dynamic semantics in *monadic* style in order to streamline the handling of errors [37]. The monadic presentation of computations that may fail consists in so-called "optional" values: either an actual value tagged with **ok**, or **error** to denote an error. The sequential composition is given in a do block (e.g. $\mathrm{do}\{A; B; C\}$). Errors are transparently propagated through such sequences: the evaluation returns **ok** v if all steps in a do block (e.g. A, B, and C) evaluate successfully, or **error** if one step in the sequence evaluates to **error**.

There are four categories of evaluations : $[\![Q]\!]_{\mathsf{D}}$ to reduce queries, $[\![e]\!]_{\eta}$ to reduce expressions, $[\![\beta]\!]_{\eta}$ to reduce aliased expressions, and $[\![\theta]\!]_{\eta}$ to reduce boolean expressions. The evaluation of a query is parametrized by a database D, which maps relation names R to tables t. A table t is a set of *rows* r, where a row r is an ordered list of values v.[4] The other categories of evaluation are parametrized by an environment η, which maps column names N to values v. Intuitively, this environment is used to extract the value associated with a column name for a given row. For instance, consider a relation of persons $\mathsf{P}(\mathsf{Name} \mapsto \mathsf{String}, \mathsf{Age} \mapsto \mathbb{Z})$, and its table $\{(\texttt{'Bob'}, 10), (\texttt{'Alice'}, 20)\}$. Then, for the first row, $\eta(\text{Name}) = \texttt{'Bob'}$ and $\eta(\text{Age}) = 10$.

Queries. The basic case is Rule (RR), which evaluates the name of a relation yielding the table associated to that name in database D. Rule (RO_{S}) evaluates set operations by first reducing the subqueries and then combining the resulting tables. Rule (Rπ) starts by evaluating subquery Q. If the result is successful (a table t), then for each row r in t, we try to project columns as dictated by β. To do this, we evaluate β under environment $\eta^{\mathsf{r}}_{\ell(Q)}$ (similarly to [20]). This environment is formed by matching corresponding column names of Q with the values of r. The list of column names of a query Q is obtained using function $\ell(Q)$, and defined inductively in the supplementary material

Finally, as the evaluation of aliased expressions can also produce errors, we lift the set of optional rows S to an optional list of rows using the $\lceil\cdot\rceil$ function defined as:

$$\lceil \mathsf{S} \rceil = \begin{cases} \mathbf{error} & \mathbf{error} \in \mathsf{S} \\ \mathbf{ok}\ \{r | (\mathbf{ok}\ r) \in \mathsf{S}\} & \text{otherwise} \end{cases}$$

Rule (Rσ) follows a similar approach by first reducing the subquery Q. If the result is successful (yielding table t), we proceed to test the reduction of the condition θ for each row. We employ a strategy akin to that of (Rπ), but in this case, the resulting set of Booleans is unused (binding the resulting in variable "_"). This way, in the third instruction, we can be confident that the evaluation of the conditions does not result in errors. Finally, we filter the rows from table t that satisfy the given condition.

[4]The supplementary material uses multi-sets instead.

$$\boxed{T \vdash e : \tau}$$

$$(\mathrm{T}v)\ \frac{}{T \vdash v : ty(v)}$$

$$(\mathrm{T}N)\ \frac{(N \mapsto \tau) \in T}{T \vdash N : \tau} \qquad (\mathrm{T}{::})\ \frac{T \vdash e : \tau' \qquad e : \tau' \rightsquigarrow \tau}{T \vdash (e :: \tau) : \tau}$$

$$(\mathrm{T}\mathsf{0})\ \frac{\begin{array}{c} T \vdash e_1 : \tau_1 \qquad T \vdash e_2 : \tau_2 \qquad \mathsf{0} \in \mathsf{0}_\mathsf{A} \cup \mathsf{0}_\mathsf{C} \\ resolve(e_1, \tau_1, e_2, \tau_2, ty(\mathsf{0})) = \tau_3 \times \tau_4 \to \tau_5 \end{array}}{T \vdash e_1\ \mathsf{0}\ e_2 : \tau_5}$$

$$(\mathrm{T}\mathsf{0}_\mathsf{B})\ \frac{T \vdash \theta_1 : \mathbb{B} \qquad T \vdash \theta_2 : \mathbb{B}}{T \vdash \theta_1\ \mathsf{0}_\mathsf{B}\ \theta_2 : \mathbb{B}} \qquad (\mathrm{T}\neg)\ \frac{T \vdash \theta : \mathbb{B}}{T \vdash \neg\theta : \mathbb{B}}$$

$$\boxed{T \vdash \beta : T'}$$

$$(\mathrm{T}\beta)\ \frac{\overline{T \vdash e : \tau} \qquad unique(\overline{N})}{T \vdash \overline{e\ \mathsf{as}\ N} : \overline{N \mapsto \tau}}$$

$$\boxed{\Gamma \vdash Q : T}$$

$$(\mathrm{T}R)\ \frac{\Gamma(R) = T}{\Gamma \vdash R : T} \qquad (\mathrm{T}\pi)\ \frac{\Gamma \vdash Q : T \qquad clean(T) \vdash \beta : T'}{\Gamma \vdash \pi_\beta(Q) : T'}$$

$$(\mathrm{T}\sigma)\ \frac{\begin{array}{c}\Gamma \vdash Q : T \\ T \vdash \theta : \mathbb{B}\end{array}}{\Gamma \vdash \sigma_\theta(Q) : T} \qquad (\mathrm{T}\times)\ \frac{\begin{array}{c}\ell(Q_1) \cap \ell(Q_2) = \emptyset \\ \Gamma \vdash Q_1 : T_1 \qquad \Gamma \vdash Q_2 : T_2\end{array}}{\Gamma \vdash Q_1 \times Q_2 : T_1, T_2}$$

$$(\mathrm{T}\mathsf{0}_\mathsf{S})\ \frac{\begin{array}{c}\Gamma \vdash \pi_{\beta_1}(Q_1) : T_1 \qquad \Gamma \vdash \pi_{\beta_2}(Q_2) : T_2 \\ biconv^*(\beta_1, T_1, \beta_2, T_2) = T \qquad \mathsf{0}_\mathsf{S} \in \{\cup, \cap, -\}\end{array}}{\Gamma \vdash \pi_{\beta_1}(Q_1)\ \mathsf{0}_\mathsf{S}\ \pi_{\beta_2}(Q_2) : T}$$

Figure 4: Type System of RAFT.
(abstract operators highlighted in gray)

Expressions. Rule (Ras) evaluates a single aliased expression by evaluating the subexpression e and disregarding the name N. Rule (Rβ) applies when we are evaluating multiple aliased expressions. Initially, it recursively reduces the sublist to a row r, and then the head of the list to a value v, resulting in a new row r, v.

Rule (RN) successfully evaluates a column name N to its corresponding value in η. Rule (Rv) successfully evaluates values to themselves. Rule (Rv ::) attempts to cast a value into a value of a different type using the *cast* function. For instance, in SQLite, the expression `CAST('hi' as INT)` evaluates to 1, whereas in PSQL is not defined. If the function is defined for the given value and type, the resulting value is returned; otherwise, an error is raised. Rule (Re::) applies to subexpressions that are not already values. It first reduces the subexpression to a value, and then casts the value using rule (Rv::).

Rules (R$\mathsf{0}_\mathsf{B}$), (R$\neg$), and (R$\mathsf{0}$) operate in a similar fashion. First, each subexpression is reduced, then the resulting values are combined using the specific operation at hand. For arithmetic and comparison operations, the exact operation is performed using the *apply* operation. For instance, in PSQL, the expression `'0'` < 1 evaluates to true, whereas in SQLite, it yields false.

4.4 Type System

A type system automatically checks a query or an expression for type errors before its evaluation, ensuring that such errors never occur at runtime. The type system of RAFT is presented in Figure 4, and is composed of three categories of typing rules, for general and aliased expressions, and for queries. Each typing rule attempts to assign a type to a query or expression and is written as an inference rule of the form $\frac{A\ldots}{B}$, where $A\ldots$ are the premises and B is the conclusion. This notation is equivalent to the implication $A \ldots \Rightarrow B$.

Expressions. Judgment $T \vdash e : \tau$ denotes that expression e has type τ under relation type T.

Rule (TN) assigns the type τ to the column name N if N is mapped to τ in the relation type T.

Rule (T::) assigns the type τ to an ascription $\tau :: e$ if the following conditions are met: (1) the expression e must be well-typed for some τ'; (2) the explicit cast of e to τ is checked to rule out explicit casts that are known to fail at runtime, using the $\cdot : \cdot \rightsquigarrow \cdot$ relation. For example, the attempt to cast `'hi'` to $\mathbb{Z}$ (`'hi'` :: $\mathbb{Z}$) is rejected in PSQL, while casting `'1'` to $\mathbb{Z}$ is accepted in both PSQL and SQLite.

Rule (Tv) assigns types to values based on the specific database engine. We achieve this by employing the *ty* operator. For instance, in PSQL, integers are typed as $\mathbb{Z}$ and literal strings as ?. However, in SQLite, both are typed as ?.

Operations are typed using rules (T$\mathsf{0}$) and (T$\mathsf{0}_\mathsf{B}$) and (T$\neg$). Rule (T$\mathsf{0}$) is responsible for both arithmetic and comparison operations. It typechecks the sub-expressions e_1 and e_2, which results in obtaining their respective types, τ_1 and τ_2. Next, the rule searches for the best candidate type signature for the given operation, taking into consideration that an operation might be overloaded with multiple types. This selection process is captured by the *resolve* and *ty*() operations. If a single candidate function type is identified, the expression is typed using the codomain. If more than one candidate type is found, the expression is considered ill-typed. For instance, in the case of the query `SELECT 1+1 FROM P` both PSQL and SQLite choose numeric addition ($\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$). However, when dealing with strings (e.g. `SELECT 'a' + 'b' FROM P`) PSQL rejects the query because it cannot not choose a best candidate operator. On the other hand, SQLite instantiates the query with numeric addition, implicitly casting both string arguments to integers. Finally, rules (T$\mathsf{0}_\mathsf{B}$) and (T$\neg$) type boolean operations.

Rule (Tβ) is used to type an aliased expression β. For simplicity, we adopt the notation $\overline{A}$ to denote a list of elements A. For instance, $\overline{e\ \mathsf{as}\ N}$ denotes the list e_1 as N_1, ..., e_2 as N_2, and $\overline{e}$ and $\overline{N}$ denote $e_1, \ldots, e_n$ and $N_1, \ldots, N_n$ respectively. Every expression e_i is typed under T, yielding a type τ_i. Subsequently, the type of the aliased expression as a whole is a relation type, where each name N_i is mapped to the type τ_i of their corresponding subexpression e_i, represented as $\overline{N \mapsto \tau}$.

Queries. Judgment $\Gamma \vdash Q : T$ represents that query Q has relation type T under schema Γ. Rule (TR) assigns a relation type to relation name R according to schema environment Γ.

Rule (Tπ) first recursively typechecks the subquery Q yielding type T. Then the projection is assigned the same type of the aliased expression β, which is typed under relation type *clean*(T). Recall, that the *clean* operator performs a transformation on a relation type, removing unknown occurrences. For instance, in PSQL, `SELECT '1' + 1 FROM P` runs successfully, but `SELECT C + 1 FROM (SELECT '1' AS C) B` is rejected statically. The reason for this is that in the first query `'1'` has type unknown ?, which can be cast to integer, whereas in the second query, the unknown type ? is transformed to String disallowing the cast to integer.

Rule (Tσ) first typechecks the subquery, resulting in a relation type T. Then, the condition must be successfully typed as Boolean under the context of T (considering that the condition may reference columns from the subquery). Finally, the selection operation is assigned the same type as the subquery T.

Typechecking query union, intersection, and difference (just set operations for simplicity) requires special attention. Usually, it is assumed that the names, cardinality, and types of the columns in the subqueries match. In practice, the column names do not need to match, but cardinality and types must align. To achieve this, many engines perform implicit casts between the columns of the subqueries to align their types. In particular, string literals are analyzed to check the plausibility of casts. For instance in PSQL, `(SELECT '1.1' AS A FROM P) INTERSECT (SELECT 1.1 AS A FROM P)` runs successfully, resulting in a non-empty result, whereas `(SELECT CAST(Age AS TEXT) AS A FROM P) INTERSECT (SELECT Age FROM P)` is rejected before execution. To deal with these special cases, and without loss of generality, we require that both subqueries within a set operation must be projections in order to verify column casts. For instance, `(SELECT '1.1' AS A FROM P) UNION (SELECT 1.1 AS A FROM P)` is encoded as $\pi_{\texttt{'1.1'}\ \mathsf{as}\ \mathsf{A}}(P) \cup \pi_{1.1\ \mathsf{as}\ \mathsf{A}}(P)$.

Rule (T$\mathsf{0}_{\mathsf{S}}$) deals with set operations over projections of lists of aliased expressions. First, it typechecks both projections. Second, it examines whether the lists of aliased expressions can be implicitly cast between each other, taking their relation types into account. Finally, if this cast is feasible, the target relation type is captured and used to typecheck the whole set operation. To check casts between list of aliased expressions, we use the *biconv** operation defined as follows:

$$biconv^*(\overline{e\ \mathsf{as}\ N}, \overline{N \mapsto \tau}, \overline{e'\ \mathsf{as}\ N'}, \overline{N' \mapsto \tau'}) = \overline{N \mapsto biconv(e,\tau,e',\tau')}$$

This operation returns a relation type, where each name N is mapped to the application of *biconv* over each pair of expressions and their types.[5] This partial function checks casts and, if possible, returns one of the two types as a result. The expressions are provided to the function to rule out casts that are known to fail during evaluation. For instance, PSQL accepts query `(SELECT '1' AS A, 1 AS B FROM R) UNION (SELECT 2 AS A, '2' AS B FROM R)`, as both `'1'` and `'2'` can be cast to integers; but rejects `(SELECT '1.1' AS A FROM R) UNION (SELECT 1 AS A FROM R)` as `'1.1'` cannot be implicitly cast to an integer (note that implicit casts from integers to strings are not allowed).

Finally, rule (T$\times$) types cross products using the concatenation of the relation types of both subqueries. This is under the condition that the sets of names in each subquery are disjoint.

4.5 Cast Insertion

Recall from Figure 1, that to avoid the complexity of dealing with implicit casts during runtime, before execution, we transform each implicit cast to an explicit cast. For instance, in our model, a PSQL query `SELECT '1' + 1 AS A FROM R` is transformed to `SELECT CAST('1' AS INT) + 1 AS A FROM R`, which in RAFT corresponds to the elaboration from $\pi_{\texttt{'1'}+1\ \mathsf{as}\ \mathsf{A}}(R)$ to $\pi_{(\texttt{'1'}::\mathbb{Z})+1\ \mathsf{as}\ \mathsf{A}}(R)$.

[5]We select N, the name of the left subquery, as it is a more commonly adopted practice in various database engines.

$$\boxed{T \vdash e : \tau \rightsquigarrow e'} \qquad (\mathrm{E}v)\ \frac{T \vdash v : \tau}{T \vdash v : \tau \rightsquigarrow v :: \tau}$$

$$(\mathrm{E}\mathsf{0})\ \frac{\begin{array}{cc} T \vdash e_1 : \tau_1 \rightsquigarrow e_1' & T \vdash e_2 : \tau_2 \rightsquigarrow e_2' \\ \mathsf{0} \in \{<,=,+\} & resolve(e_1,\tau_1,e_2,\tau_2,ty(\mathsf{0})) = \tau_3 \times \tau_4 \longrightarrow \tau_5 \end{array}}{\begin{array}{l} T \vdash e_1\ \mathsf{0}\ e_2 : \tau_4 \rightsquigarrow \\ \qquad insert(insert(e_1',\tau_3,\mathsf{0})\ \mathsf{0}\ insert(e_2',\tau_4,\mathsf{0}),\tau_5,\mathsf{0}) \end{array}}$$

$$\boxed{\Gamma \vdash Q : T \rightsquigarrow Q'}$$

$$(\mathrm{T}\mathsf{0}_{\mathsf{S}})\ \frac{\begin{array}{cc} \Gamma \vdash \pi_{\beta_1}(Q_1) : T_1 \rightsquigarrow \pi_{\beta_1'}(Q_1') & \Gamma \vdash \pi_{\beta_2}(Q_2) : T_2 \rightsquigarrow \pi_{\beta_2'}(Q_2') \\ biconv^*(\beta_1,T_1,\beta_2,T_2) = T & \mathsf{0}_{\mathsf{S}} \in \{\cup,\cap,-\} \\ e_1 = insert^*(\beta_1',T,\mathsf{0}_{\mathsf{S}}) & e_2 = insert^*(\beta_2',T,\mathsf{0}_{\mathsf{S}}) \end{array}}{\Gamma \vdash \pi_{\beta_1}(Q_1)\ \mathsf{0}_{\mathsf{S}}\ \pi_{\beta_2}(Q_2) : T \rightsquigarrow \pi_{e_1}(Q_1')\ \mathsf{0}_{\mathsf{S}}\ \pi_{e_2}(Q_2')}$$

Figure 5: RAFT **Cast Insertion (excerpt). (abstract operators highlighted in gray)**

An excerpt of the elaboration rules are presented in Figure 5; the complete rules can be found in the supplementary material. They are *type directed*, meaning that (1) we only elaborate well-typed terms, and (2) we use type information during elaboration. Like for typing, elaboration rules are defined inductively and grouped in three categories: for general and aliased expressions, and for queries. Most elaboration rules directly follow their corresponding typing rule. Judgment $T \vdash e : \tau \rightsquigarrow e'$ represents that expression e typed as τ under relation type T is elaborated to e'. Judgment $\Gamma \vdash Q : T \rightsquigarrow Q'$ is defined analogously.

Rule (Ev) inserts an explicit cast to the type of each value. This is especially relevant for engines like SQLite where some values need to be tagged as unknown. For example, the expression `'0'` < 1 evaluates to 0 (which represents false in that engine). To facilitate this special case, the expression is elaborated to `'0'` $:: ? < 1 :: ?$.

Rule (E$\mathsf{0}$) introduces explicit casts based on the operation and the best candidate type. Specifically, we insert casts for both operands to match their expected corresponding domain types and also insert a cast in the result of the operation to the expected codomain type. Certain engines, like SQLite, do not insert explicit casts for comparison operations. To achieve this, we use the *insert* operator.

Rule (E$\pi\beta$) elaborates set operations by inserting casts to the output type of *biconv**. This is done using the *insert** operation defined as follows:

$$insert^*(\overline{e\ \mathsf{as}\ N}, \overline{N \mapsto \tau}, \mathsf{0}_{\mathsf{S}}) = \overline{insert(e,\tau,\mathsf{0}_{\mathsf{S}})\ \mathsf{as}\ N}$$

For instance, for PSQL, query `(SELECT '1' FROM R) INTERSECT (SELECT 1 FROM R)` is elaborated to `(SELECT CAST('1' AS INT) FROM R) INTERSECT (SELECT CAST(1, INT) FROM R)`.

4.6 Translation from SQL to RAFT

Figure 6 presents a subset of the translation rules from SQL to RAFT. These rules are defined inductively and divided into three

$$\boxed{B \rightsquigarrow_b \beta} \qquad (\mathrm{Tr}E)\,\frac{E\ \texttt{AS}\ \mathit{alias}(E) \rightsquigarrow_b e}{E \rightsquigarrow_b e}$$

$$\boxed{F \rightsquigarrow Q} \qquad (\mathrm{Tr}Fas)\,\frac{S \rightsquigarrow Q \qquad \ell(Q) = N_1, \ldots}{S\ N \rightsquigarrow \pi_{N_1 \mathsf{as} N.N_1, \ldots}(Q)}$$

Figure 6: SQL to RAFT **translation (excerpt)**

categories: translation rules for expressions, aliased columns, and queries. Most of the rules are straightforward. Rule (TrE) transforms expressions without an alias, as seen in `SELECT 1+1 FROM R`. In these cases, we utilize the $\mathit{alias}(\cdot)$ function, which generates an appropriate name for the subexpression. For example, $\mathit{alias}(1+1) =$ `"1+1"`, and $\mathit{alias}($`NAME`$) =$ `NAME`. Rule (TrFas) transforms aliased subqueries by creating a projection of the translated subquery, where each column is aliased using the table name plus the column name. The complete rules can be found in the supplementary material.

In addition, we prove that every select query S can be translated to a unique query in RAFT.

Theorem 4.1. *For any SQL query S there exists a unique relational algebra query Q such that $S \rightsquigarrow Q$.*

5 INSTANTIATING RAFT

We now demonstrate how to instantiate RAFT for two radically different database engines: PSQL and SQLite. The instantiations for other engines can be found in the supplementary material. An instantiation is achieved by providing specific definitions of the abstract operators. We obtained these definitions by exploring the documentation of each engine, and by conducting black-box analyses interacting with each engine whenever the documentation was lacking in details.

5.1 RAFT/PSQL

Figure 7 describes the RAFT/PSQL instantiation. PSQL is characterized by being a strongly-typed SQL engine, meaning that it is more conservative than the rest. In many cases, if it cannot check the feasibility of casts, it rejects the query before its execution.

Bidirectional implicit cast. Operator *biconv* is defined using the auxiliary *implicit cast* relation $e : \tau \rightsquigarrow \tau'$. An expression e of type τ can be cast to τ' if either type τ can be implicitly cast to τ' ($\tau \Rightarrow \tau'$), or if e is a value v of type unknown ? (i.e. a string) and that value can be implicitly cast to some value v' under type τ' ($\mathit{icast}(v, \tau') = v'$).

Implicit type cast $\tau \Rightarrow \tau'$ is only defined between identical types $\tau \Rightarrow \tau$, or from integers to reals $\mathbb{Z} \Rightarrow \mathbb{R}$. For instance, (`SELECT 1 FROM R`) `INTERSECT` (`SELECT 1.0 FROM R`) is accepted and evaluates to $\{1.0\}$, since $\mathbb{Z}$ (the type of `1`) can be implicitly cast to a $\mathbb{R}$ (the type of `1.0`). Implicit value cast $\mathit{icast}(v, \tau) = v'$ is defined for extracting numbers from strings, but not viceversa. Conversions from integer to string are only allowed in explicit casts. For instance, (`SELECT '-1.0' FROM R`) `INTERSECT` (`SELECT -1.0 FROM R`) is accepted and evaluates to a set containing -1.0.

Explicit cast feasibility. If the expression tested is not a value, the relation vacuously holds. Conversely, if e is a value v, it is eagerly checked whether the explicit cast to τ' fails or not, using the explicit cast relation.

Explicit cast. In addition to what an implicit cast can do, explicit casts support casts from real numbers to integers by removing

$$\boxed{\mathit{biconv}(e_1, \tau_1, e_2, \tau_2) = \tau}$$

$$\frac{e_1 : \tau_1 \rightsquigarrow \tau_2}{\mathit{biconv}(e_1, \tau_1, e_2, \tau_2) = \tau_2} \qquad \frac{e_2 : \tau_2 \rightsquigarrow \tau_1}{\mathit{biconv}(e_1, \tau_1, e_2, \tau_2) = \tau_1}$$

$$\boxed{e : \tau \rightsquigarrow \tau'} \qquad \boxed{\mathit{icast}(v, \tau) = v'}$$

$$\frac{\tau \Rightarrow \tau'}{e : \tau \rightsquigarrow \tau'} \qquad \frac{\mathit{icast}(v, \tau) = v'}{v : ? \rightsquigarrow \tau}$$

$$\mathit{icast}(\text{‘}n\text{’}, \mathbb{R}) = n \qquad \mathit{cast}(n, \mathbb{R}) = n \qquad \mathit{icast}(v, \mathit{ty}(v)) = v \qquad \mathit{icast}(\text{‘}v\text{’}, \mathit{ty}(v)) = v \quad v \neq s$$

$$\boxed{e : \tau \overset{\sim}{\rightsquigarrow} \tau'} \qquad \boxed{\mathit{cast}(v, \tau) = v'}$$

$$\frac{e \neq v}{e : \tau \overset{\sim}{\rightsquigarrow} \tau'} \qquad \frac{\mathit{cast}(v, \tau') = v'}{v : \tau \overset{\sim}{\rightsquigarrow} \tau'}$$

$$\mathit{cast}(n_1.n_2, \mathbb{Z}) = n_1 \qquad \mathit{cast}(v, \text{String}) = \text{‘}v\text{’} \quad v \neq s \qquad \mathit{cast}(v, \tau) = \mathit{icast}(v, \tau)$$

$$\boxed{\mathit{bestCandidate}(\tau_1, \tau_2, \overline{\tau_3}) = \tau}$$

$$\frac{\{m\} = \underset{i}{\arg\min}((\tau 1 - \tau_{1i}) + (\tau_2 - \tau_{2i})) \qquad \tau_1 \Rightarrow \tau_{1m} \qquad \tau_2 \Rightarrow \tau_{2m}}{\mathit{bestCandidate}(\tau_1, \tau_2, \overline{\tau_{1i} \times \tau_{2i} \longrightarrow \tau_{3i}}) = \tau_{3m} \times \tau_{4m} \longrightarrow \tau_{3m}}$$

$$\tau - \tau = 0 \quad \mathbb{Z} - \mathbb{R} = 1 \quad \text{String} - \mathbb{Z} = 1 \quad \text{String} - \mathbb{R} = 1 \quad \_ - \_ = 2$$

$$\boxed{\mathit{resolve}(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau}$$

$$\frac{\tau_1 \neq ? \qquad \tau_2 \neq ? \qquad \mathit{bestCandidate}(\tau_1, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}{\mathit{resolve}(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}$$

$$\frac{\tau_1 \neq ? \qquad \tau_2 = ? \qquad \mathit{bestCandidate}(\tau_1, \tau_1, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6 \qquad \mathit{icast}(e_2, \tau_5) = v_2}{\mathit{resolve}(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}$$

$$\frac{\tau_1 = ? \qquad \tau_2 \neq ? \qquad \mathit{bestCandidate}(\tau_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6 \qquad \mathit{icast}(e_1, \tau_4) = v_1}{\mathit{resolve}(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}$$

$$\frac{\tau_1 = ? \qquad \tau_2 = ? \qquad \mathit{bestCandidate}(\text{String}, \text{String}, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}{\mathit{resolve}(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}$$

Basic definitions

$$\mathit{ty}(n) = \mathbb{Z} \quad \mathit{ty}(r) = \mathbb{R} \quad \mathit{ty}(s) = ?$$

$$\mathit{insert}(e, \tau, 0) = e :: \tau \qquad \mathit{clean}(T) = T[\text{String}/?]$$

$$\mathit{ty}(0_C) = \{\mathbb{Z} \times \mathbb{Z} \to \mathbb{B}, \mathbb{R} \times \mathbb{R} \to \mathbb{B}, \text{String} \times \text{String} \to \mathbb{B}\}$$

$$\mathit{ty}(+) = \{\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \mathbb{R} \times \mathbb{R} \to \mathbb{R}\}$$

$$\mathit{apply}(0, v_1, v_2) = v_1\ 0\ v_2$$

Figure 7: The RAFT/PSQL **Instantiation**

the decimals, and from non-string values to strings by enclosing them in quotes. For instance, `SELECT CAST('1.1' AS DOUBLE) FROM R` is accepted and evaluates to 1.1, but `SELECT CAST('1.1' AS INT) FROM R` is rejected by the typechecker.

Overload resolution. This rule is divided in four cases. The general case arises when the types of the two expressions are not unknown. Function $bestCandidate(\tau_1, \tau_2, \overline{\tau_3})$ is used to determine the best candidate. We model this function by initially calculating the sum of the type differences between the corresponding types of the expression and the domain of each candidate. The type difference $\tau - \tau'$ quantifies the "cost" of changing type τ to τ': it yields a value of 0 if both types are identical, 1 when transitioning from an integer to a real or from a string to a number, and 2 in all other cases. If there are ties, and more than one type is selected, then the function is not defined. Furthermore, both types must satisfy the implicit cast relation with their corresponding type from the domain of the chosen best candidate. For instance, for query `SELECT 1 + 1.1 FROM R`, the + operation has two possible candidates: $\mathbb{Z}\times\mathbb{Z} \to \mathbb{Z}$, and $\mathbb{R}\times\mathbb{R} \to \mathbb{R}$. The best candidate in this case is $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, as both operands can be implicitly cast to reals.

The remaining cases are analogous. When only one type is unknown, the best candidate function is applied using the other known type in both positions. Additionally, the expression of unknown type is implicitly cast to the chosen type to rule out potential errors. Finally, if both types are unknown, they are assumed to be strings when looking for the best candidate. For example, in `SELECT '1' + 1.1 FROM R`, the best candidate type is $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, as the implicit cast from `'1'` to real is possible. On the contrary, query `SELECT '1' + '1' FROM R` is rejected by the typechecker because there is more than one candidate available (int and real versions).

Basic definitions. For the type of values, reals are typed as $\mathbb{R}$, integers to $\mathbb{Z}$, and strings literals to ?. The operator for removing unknown types replaces ? occurrences with String, while leaving other types unchanged.[6] The operator for inserting annotations adds explicit casts regardless of the operation's type. The candidate types of a given operator is represented as sets of binary function types. Lastly, the semantics of operations between values are passed to the real implementation without any modifications.

5.2 RAFT/SQLite

Figure 8 describes the RAFT/SQLite instantiation. SQLite is one of the most flexible SQL engines: type enforcement is not mandatory, and types on columns are optional. However, it attempts its best effort to perform casts without ever raising a type error at runtime. To capture this kind of flexibility, we model the type of values as ? and define all abstract operators as total functions.

Bidirectional implicit cast. Operator *biconv* yields always the unknown type for any pair of types and expressions.

Implicit value casts closely resemble explicit value casts in PSQL. The only distinction lies in the fact that the relation is always defined. Consequently, for implicit casts from strings to numbers, when the string is not a valid number, the result is the same string.

Explicit cast feasibility. It is always feasible to cast any expression of type τ to any type τ'.

[6]Notation $C[A/B]$ denotes type C where occurrences of B have been replaced by A.

$$\boxed{icast(v,\tau) = v'} \quad \ldots$$

$$\begin{aligned} icast(s,\tau) &= icast(number(s),\tau) \quad \tau \in \{\mathbb{Z},\mathbb{R}\} \\ icast(s,\tau) &= s \quad number(s) \text{ is not def.} \end{aligned}$$

$$\boxed{cast(v,\tau) = v'} \quad \ldots$$

$$\begin{aligned} cast(s,\tau) &= cast(number(nprefix(s)),\tau) \quad \tau \in \{\mathbb{Z},\mathbb{R}\} \\ cast(s,\tau) &= 0 \quad number(nprefix(s)) \text{ is not defined}, \tau \in \{\mathbb{Z},\mathbb{R}\} \end{aligned}$$

$$\tau - \tau = 0 \quad \mathbb{Z} - \mathbb{R} = 0 \quad \mathsf{String} - \mathbb{R} = 1 \quad ? - \tau = 1 \quad \_ - \_ = 2$$

Basic definitions

$$ty(v) = ? \quad dty(w :: ?) = ? \quad dty(r) = \mathbb{R} \quad dty(s) = \mathsf{String}$$

$$insert(e,\tau,\odot_C) = e \quad insert\ (e,\tau,+) = e :: \tau$$

$$ty(\odot_C) = \{? \times ? \to ?, \mathbb{R}\times\mathbb{R} \to ?, \mathsf{String} \times \mathsf{String} \to ?\}$$

$$ty(+) = \{\mathbb{R}\times\mathbb{R} \to ?\}$$

$$apply(+, v_1, v_2) = v_1 + v_2$$

$$\begin{aligned} apply(\odot_C, v_1, v_2) &= compare(|v_1'|, |v_2'|) :: ? \\ &\text{where } resolve(v_1, dty(v_1), v_2, dty(v_2), ty(\odot_C)) = \tau_1 \times \tau_2 \to \tau_3, \\ &\qquad icast(v_1,\tau_1) = v_1', icast(v_2,\tau_2) = v_2' \end{aligned}$$

$$|w :: ?| = w \quad |w| = w$$

Figure 8: The RAFT/SQLite Instantiation (excerpt)

Explicit cast. This operator is defined almost identical to implicit casts, except for the case when the expression is a string. In this case, the cast is performed by extracting the largest numeric prefix from the string and then casting it to the required number type. If there is no numeric prefix, then the cast yields 0. For instance, `SELECT CAST('12.3hi' AS INT), CAST('hi') FROM R` evaluates to $(12, 0)$.

Overload resolution. There is only one rule for overloading resolution. This rule is similar to PSQL but it yields the first best candidate found. The best candidate function does not perform implicit casts between types. Type difference now yields 0 when going from integer to real, and 2 from unknown to any other type.

Basic definitions. Statically, the type of every value is unknown. We introduce a dynamic type operator $dty(v)$ to obtain precise type information during the evaluation of comparisons. The type of a value, initially unknown, may be refined on runtime to a more precise type. In fact $1 :: ?$ does not behave as 1. Therefore the operator for removing unknown types must act as the identity function. Regarding the operator for inserting annotations, it only inserts explicit casts for arithmetic operations, while for comparison operations, it behaves as the identity function.

The dynamic semantics for comparison is involved. First, the best candidate type is searched, using the dynamic type information of the operands. Then, once the best candidate is found, both operands are implicitly cast to the corresponding types. Finally, the cast values, stripped of (potentially) casts to unknown, are compared using the $compare(w_1, w_2)$ function defined as 1 if $(dty(w_1) = dty(w_2) \land w_1 < w_2) \lor dty(w_1) < dty(w_2)$; 0 otherwise. If the dynamic types of the operands are equal, then a regular comparison operation is performed. If the types are different then the types are compared using an arbitrary hierarchy such that $\mathbb{Z} = \mathbb{R} < \mathsf{String}$.

6 PROPERTIES

Based on the core relational algebra of RAFT, we can establish metatheoretical results for each engine considered, consisting of multiple lemmas and theorems regarding queries and their evaluation. Specifically, we can articulate the formal distinctions among various engines. This aids us in better comprehending the process of transforming queries from one engine to another.

To state these theorems, we need several new definitions, in particular a way to typecheck rows r, tables t and databases D:

$$\frac{\vdash v : \tau}{\vdash v : (N \mapsto \tau)} \qquad \frac{\vdash r : T \qquad \vdash v : \tau}{\vdash r, v : T, (N \mapsto \tau)} \qquad \frac{}{\vdash \cdot : T}$$

$$\frac{\forall r \in t. \vdash r : T}{\vdash t : clean(T)} \qquad \frac{\forall R \in dom(\mathsf{D}). \vdash \mathsf{D}(R) : \Gamma(R)}{\vdash \mathsf{D} : \Gamma}$$

A row is well-typed if every value is typed to its corresponding type in T. A table is typed T if every row is typed as T. An empty row is typed to any relation type T. A database is typed Γ, if every relation in D is typed to its corresponding type in Γ. The proofs require some properties about the implementation of abstract operators:

R1: Every operator must be deterministic.
R2: $biconv(e_1, \tau_1, e_2, \tau_2)$ yields either τ_1 or τ_2.
R3: $resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3})$ must be contained in $\overline{\tau_3}$.
R4: If $cast(v, \tau) = v'$, then the (cleaned) type of v' must be τ.
R5: If $resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6$, then $[\![e_1 :: \tau_4]\!]_\eta \neq$ **error** and $[\![e_2 :: \tau_5]\!]_\eta \neq$ **error**.
R6: If $biconv(e_1, \tau_1, e_2, \tau_2) = \tau$, then $[\![e_1 :: \tau]\!]_\eta \neq$ **error** and $[\![e_2 :: \tau]\!]_\eta \neq$ **error**.
R7: If $resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6$ then $e_1 : \tau_1 \rightsquigarrow \tau_4$ and $e_2 : \tau_2 \rightsquigarrow \tau_5$.
R8: If $biconv(e_1, \tau_1, e_2, \tau_2) = \tau$ then $e_1 : \tau_1 \rightsquigarrow \tau$ and $e_2 : \tau_2 \rightsquigarrow \tau$.
R9: Either $clean(T) = T[\text{String}/?]$ or $clean(T) = T$.

Assuming R1, R2, R3, and R4, any instantiation of RAFT is *type safe*, meaning that well-typed queries either reduce to a table or raise a (controlled) type error. In other words, evaluation of well-typed queries does not get stuck. For instance, an ill-type query, or `SELECT Foo FROM P`, where `Foo` is not defined in `P`, gets stuck as $[\![\texttt{Foo}]\!]_\eta$ and does not evaluate.

Theorem 6.1 (Type Safety). *$\forall T\ Q\ \mathsf{D}$, if $\Gamma \vdash Q : T$ and $\vdash \mathsf{D} : \Gamma$ then $(\exists t, [\![Q]\!]_{\mathsf{D}} = t) \vee ([\![Q]\!]_{\mathsf{D}} =$ **error**$)$.*

Assuming also R4, a stronger theorem called *type soundness* states that, in addition to type safety, the resulting table indeed has the type of the query.

Theorem 6.2 (Type Soundness). *$\forall T\ Q\ \mathsf{D}$, if $\Gamma \vdash Q : T$ and $\vdash \mathsf{D} : \Gamma$ then $(\exists t, T'.[\![Q]\!]_{\mathsf{D}} = t, \vdash t : T'$ and $clean(T') = clean(T)) \vee ([\![Q]\!]_{\mathsf{D}} =$ **error**$)$.*

The use of $clean(\cdot)$ is exclusively for PSQL, and for cases such as `SELECT CAST('hi' as String) as A from R`. This query of type $A \mapsto$ String evaluates to $\{$`'hi'`$, ...\}$, typed as $A \mapsto ?$ (literal strings are typed ?), but $clean(\text{String}) = clean(?) = \text{String}$.

Type safety is satisfied by every engine we consider, but type soundness is satisfied by all except SQLite. This is because SQLite does not satisfy R3. For instance, in SQLite, `SELECT CAST(1 as INT) AS A FROM R` has type $A \mapsto \mathbb{Z}$, but its evaluation is typed $A \mapsto$?. Also, SQLite permits storing string values in integer columns.

Moreover, PSQL, MySQL and SQLite satisfy a theorem that states that if the programmer does not use any explicit cast in a well-typed query, then the query evaluates without error. To state this theorem we use the cast-free metafunction $\mathsf{CF}(Q)$, which is defined when Q does not have explicit casts of the form $e :: \tau$ (definition in supplementary material).

Theorem 6.3 (Cast-Free Queries are Total). *If $\mathsf{CF}(Q)$, $\vdash \mathsf{D} : \Gamma$ and $\Gamma \vdash Q : T \rightsquigarrow Q'$ then $[\![Q']\!]_{\mathsf{D}} \neq$ **error**.*

This theorem requires R1, R5, R6 and R9. Both MSSQL and Oracle do not satisfy this property. Specifically, MSSQL does not satisfy R5 and R6, and Oracle does not satisfy R5. To illustrate why, let us consider table $R = \{($`'1'`$)\}$, schema $R \mapsto (A \mapsto \text{String})$, and query `SELECT A + 1 FROM R`. This query does not typecheck in PSQL, and evaluates successfully in other engines. But with one more row to R: $\{($`'1'`$), ($`'hi'`$)\}$, the same query evaluates to a runtime error in MSSQL and Oracle.

Regarding cast insertion, translated queries preserve types, and the translation is unique:

Theorem 6.4 (Cast insertion is a type-preserving function). *If $\Gamma \vdash Q : T$ then there exists a unique Q' such that $\Gamma \vdash Q : T \rightsquigarrow Q'$ and $\Gamma \vdash Q' : T$.*

This theorem requires R1, R2, R4, R7, R8 and R9, and is satisfied by every engine we considered.

7 VALIDATION

To validate the adequacy of the formalism, we develop PyRAFT, an implementation of RAFT in Python, and create five instances for each engine. We generate multiple random queries and verify if the results from the actual engine match those obtained from the prototype.

PyRAFT is composed of four main modules: a generator for creating random queries, a parser that transforms SQL queries into relational algebra queries, a type checker, and an evaluator responsible for evaluating relational algebra queries. Contrary to the formalism presented in previous sections, the evaluator represents tables as unordered lists instead of sets, closely resembling the bag semantics found in the supplementary material.

For testing purposes, we constructed three tables, each comprised of three columns containing strings, integers, and real numbers. Keeping the tables small accelerates the validation process, as our focus lies not on performance but on behavioral differences to validate the RAFT formalism. Additionally, we generated random expressions consisting of arithmetic and comparison operations.

We generate a total of 100, 000 random SQL queries for each engine, successfully confirming that our design aligns with the behavior of each individual engine. This process is challenging when dealing with engine-specific query optimizations. In particular, sometimes PyRAFT reports an error while the engine returns a table. This discrepancy occurs due to avoidance of executing certain subexpressions or subqueries that are prone to failure. For this reason, we divided the validation in two categories: a *termination-insensitive validation*, and a *termination-sensitive validation.*

The termination-insensitive validation approach involves verifying that, if the evaluation of a query in PyRAFT and in the engine result in tables, then these tables must be equivalent. In PyRAFT, the query generation is parameterized by the engine due to subtle discrepancies between engines. For instance, MSSQL and Oracle lack a Boolean type and represent Booleans using integers. To avoid floating number precision mismatches, real numbers are represented as decimals in both PyRAFT and the engines.[7] Note that comparing real numbers using a notion of closeness might be feasible, but it presents a greater challenge when these results are then cast to strings. Finally, in MySQL, we had to cast some operands of arithmetic operators to decimal to avoid precision issues. For instance, query `select 'hi' + 5.1 - 9 FROM R` evaluates in PyRAFT to -3.9, but to $\{-3.9000000000000004\}$ in MySQL.

The termination sensitive validation approach is a stronger result. It involves verifying that, if the evaluation of a query in PyRAFT yields a table, then evaluation of the query in the engine results in an equivalent table. Furthermore, if a query in PyRAFT reports an error, then the query in the engine also reports an error. It's important to note that sometimes distinguishing between errors resulting from type checking or evaluation solely by inspecting the engine's output might not be feasible. Consequently, if PyRAFT reports a type error (either statically or dynamically), we verify if the real engine throws any kind of errors.

To achieve this stronger validation, we had to perform several simplifications on the generation of queries. This was necessary because the engine's query optimizations prevent some sub-expressions from being evaluated. Among the troublesome optimization we find: (1) expressions within subqueries are not evaluated if the selection columns do not depend on the subquery's columns; (2) unpredictable order of evaluation of logical conjunctions and short-circuit implementation; (3) the engine predicts that intersections are empty by examining only the first column of both tables. A detailed list of simplifications is described in the supplementary material.

8 RELATED WORK

Traditionally SQL has been implemented with some sort of either static typing or syntactic checking, though the issue of type errors and type disciplines has received little attention. Nonetheless, there are two lines of works that relates to our problem. In the Database literature, the consideration of corner cases such as NULLs and dynamic generated queries involved typing issues. Also, some engines, like SQLLite have "flexible" type systems. In Programming Languages type system is a central topic, although SQL and databases have received little attention. Both areas have been source of inspiration and techniques for our work.

Classical Database literature. There are many works formalizing SQL [9–11, 28]. Guagliardo and Libkin [20] developed a comprehensive formal semantics for SQL whose core we follow here. Following the classic framework in the area, they assume that all comparisons and operations are applied to arguments of the right types. Therefore essentially they do not deal with typing issues. Regarding errors in SQL, based on previous work [4, 33, 39], Taipalus et al. [35] review SQL errors to build a unified error categorization. In further work, Taipalus et al. [34] compared the error messages of the four most popular relational database management systems (MySQL, Oracle, PostgreSQL, and SQL Server) in terms of error message effectiveness, effects, and usefulness, and error recovery confidence. Our work does not deal with error messages, but instead with detecting errors. Finally, regarding formalization, Benzaken et al. [6] proposed the first mechanically verified compiler (DBCert, using Coq) for SQL queries. Ricciotti and Cheney [31] complemented and deepended the work of Guagliardo and Libkin [20] by making the notions of their semantics and proof precise and formal using Coq. These works assume that every type is matched and therefore there are no implicit casts.

Flexible typing databases. A distinctive example is SQLite, the most widely deployed database engine [22], that enjoys flexible typing. Data of any type may be stored in any column of an SQLite table (except an INTEGER PRIMARY KEY column, in which case the data must be integral) and columns can be declared without a data type [13]. SQL queries are viewed as strings and little error checking is done for dynamically generated SQL query strings. Wassermann et al. [38] proposed a static program analysis technique to verify that dynamically generated query strings do not contain type errors. Similar to RAFT, they employ a type system to reject dynamically generated queries. However, their focus does not lie in the formalism of dynamic semantics or casts, and the evaluation uses the grammar of Oracle.

Strongly-typed queries. The development of programming language libraries and tools for type-checking queries has been extensively explored [5, 14–17, 25–27, 32]. However, the formalization of implicit and explicit type casts has not been addressed. Additionally, these studies lack a practical exploration of the varied behaviors induced by typing in industry-standard database engines. From a formal perspective, significant progress has been made in the area of type inference for relational algebra [8, 29, 36] and SQL [12, 24]. In RAFT, we presume the existence of a typed schema, and consider the implementation of type inference as an area for future exploration.

9 CONCLUSION

In this paper, we identified some discrepancies in behavior regarding the handling of types both statically and dynamically in current SQL engines. This makes difficult to port queries from one engine to another. As a first step to address this problem, we presented a formal framework for a typed relational algebra, named RAFT, with support for implicit and explicit type casts. Through RAFT, we can formally understand the behavior of different database engines; we provided five instantiations and described those of PostgreSQL and SQLite. Our framework, highlights the necessary requirement for any concrete instantiation to satisfy formal properties such as type safety and soundness, among others. We validated our formalization via prototype implementations, comparing thousands of randomly generated queries. The immediate future work we foresee is to study discrepancies under query optimizations performed by many practical engines, and to improve error messages.

REFERENCES

[1] 2020. Converting MySQL to PostgreSQL. https://en.wikibooks.org/wiki/Converting_MySQL_to_PostgreSQL.

[2] 2023. Converting MySQL to Postgre online. https://en.wikibooks.org/wiki/Converting_MySQL_to_PostgreSQL.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/

[7] In Python, $1.9 - 0.1 = 1.7999999999999998$

[4] Alireza Ahadi, Vahid Behbood, Arto Vihavainen, Julia Prior, and Raymond Lister. 2016. Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and Its Application to Predicting Students' Success. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 401–406. https://doi.org/10.1145/2839509.2844640

[5] Lennart Augustsson and Mårten Ågren. 2016. Experience Report: Types for a Relational Algebra Library. *SIGPLAN Not.* 51, 12 (sep 2016), 127–132. https://doi.org/10.1145/3241625.2976016

[6] Véronique Benzaken, Évelyne Contejean, Mohammed Houssem Hachmaoui, Chantal Keller, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2022. Translating Canonical SQL to Imperative Code in Coq. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 83 (apr 2022), 27 pages. https://doi.org/10.1145/3527327

[7] HL Bhandari and R Chitrakar. 2020. Comparison of Data Migration Techniques from SQL Database to NoSQL Database. *J Comput Eng Inf Technol 9* 6 (2020), 2.

[8] Peter Buneman and Atsushi Ohori. 1996. Polymorphism and Type Inference in Database Programming. *ACM Trans. Database Syst.* 21, 1 (mar 1996), 30–76. https://doi.org/10.1145/227604.227609

[9] Stefano Ceri and Georg Gottlob. 1985. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Transactions on Software Engineering* SE-11 (1985), 324–345. https://api.semanticscholar.org/CorpusID:22717180

[10] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:12408033

[11] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2016. HoTTSQL: proving query rewrites with univalent SQL semantics. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016). https://api.semanticscholar.org/CorpusID:644867

[12] Dario Colazzo and Carlo Sartiani. 2011. Precision and Complexity of XQuery Type Inference. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming* (Odense, Denmark) *(PPDP '11)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/2003476.2003490

[13] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* 15 (2022), 3535–3547. https://api.semanticscholar.org/CorpusID:252066674

[14] Carl Gould, Zhendong Su, and Premkumar T. Devanbu. 2004. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum (Eds.). IEEE Computer Society, 697–698. https://doi.org/10.1109/ICSE.2004.1317494

[15] The Doobie Development Group. 2019. Typechecking Queries. https://tpolecat.github.io/doobie/docs/06-bChecking.html.

[16] The Kysely Development Group. 2021. Kysely. https://kysely.dev/.

[17] The PgTyped Development Group. 2020. PgTyped. https://github.com/adelsz/pgtyped.

[18] The PostgreSQL Global Development Group. 1996. PostgreSQL Documentation. https://www.postgresql.org/docs/current/typeconv-boverview.html.

[19] The SQLines Development Group. 2010. SQLines. http://www.sqlines.com/online.

[20] Paolo Guagliardo and Leonid Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *Proc. VLDB Endow.* 11, 1 (sep 2017), 27–39. https://doi.org/10.14778/3151113.3151116

[21] Stephanie W. Haas. 2004. Erik Peter Bansleben . Database Migration : A Literature Review and Case Study. https://api.semanticscholar.org/CorpusID:17518212

[22] D. Richard Hipp. [n.d.]. Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html

[23] Shafaq Khan, Ajish Kalia, Haleh Mehdipour Dastjerdi, and Nishara Nizamuddin. 2023. Automated Tool for NoSQL to SQL Migration. In *Proceedings of the 7th International Conference on Information Systems Engineering* (, Charleston, SC, USA,) *(ICISE '22)*. Association for Computing Machinery, New York, NY, USA, 20–23. https://doi.org/10.1145/3573926.3573931

[24] Weisheng Lin. 2004. *Type inference in SQL*. Ph.D. Dissertation. Concordia University.

[25] Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)* (2010).

[26] MIT. 2019. TS-SQL-Query. https://ts-bsql-bquery.readthedocs.io/.

[27] Claudia Mónica Necco and J Nuno Olivera. 2005. Toward generic data processing. In *XI Congreso Argentino de Ciencias de la Computación*.

[28] M. Negri, G. Pelagatti, and L. Sbattella. 1991. Formal Semantics of SQL Queries. *ACM Trans. Database Syst.* 16, 3 (sep 1991), 513–534. https://doi.org/10.1145/111197.111212

[29] Atsushi Ohori and Peter Buneman. 1988. Type Inference in a Database Programming Language. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming* (Snowbird, Utah, USA) *(LFP '88)*. Association for Computing Machinery, New York, NY, USA, 174–183. https://doi.org/10.1145/62678.62700

[30] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[31] Wilmer Ricciotti and James Cheney. 2022. A Formalization of SQL with Nulls. *J. Autom. Reason.* 66, 4 (nov 2022), 989–1030. https://doi.org/10.1007/s10817-b022-b09632-b4

[32] Alexandra Silva and Joost Visser. 2006. Strong Types for Relational Databases. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell* (Portland, Oregon, USA) *(Haskell '06)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1159842.1159846

[33] John B. Smelcer. 1995. User Errors in Database Query Composition. *Int. J. Hum.-Comput. Stud.* 42, 4 (apr 1995), 353–381. https://doi.org/10.1006/ijhc.1995.1017

[34] Toni Taipalus, Hilkka Grahn, and Hadi Ghanbari. 2021. Error messages in relational database management systems: A comparison of effectiveness, usefulness, and user confidence. *Journal of Systems and Software* 181 (2021), 111034. https://doi.org/10.1016/j.jss.2021.111034

[35] Toni Taipalus, Mikko Siponen, and Tero Vartiainen. 2018. Errors and Complications in SQL Query Formulation. *ACM Trans. Comput. Educ.* 18, 3, Article 15 (aug 2018), 29 pages. https://doi.org/10.1145/3231712

[36] Jan Van den Bussche and Emmanuel Waller. 2002. Polymorphic Type Inference for the Relational Algebra. *J. Comput. System Sci.* 64, 3 (2002), 694–718. https://doi.org/10.1006/jcss.2001.1812

[37] Philip Wadler. 1992. Comprehending Monads. *Mathematical Structures in Computer Science* 2 (1992), 461–493.

[38] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. 2007. Static Checking of Dynamically Generated Queries in Database Applications. *ACM Trans. Softw. Eng. Methodol.* 16, 4 (sep 2007), 14–es. https://doi.org/10.1145/1276933.1276935

[39] Charles Welty. 1985. Correcting User Errors in SQL. *International Journal of Man-Machine Studies* 22, 4 (1985), 463–477.

$\boxed{biconv(e_1, \tau_1, e_2, \tau_2) = \tau}$

$$\frac{}{biconv(e_1, \tau_1, e_2, \tau_2) = ?}$$

$\boxed{icast(v, \tau) = v'}$

$$\begin{array}{rlrl}
number(\text{‘}n\text{’}) & = n & number(\text{‘}r\text{’}) & = r \\
icast(v, ty(v)) & = v & icast(n_1.n_2, \mathbb{Z}) & = n_1 \\
icast(n, \mathbb{R}) & = n & icast(v, \text{String}) & = \text{‘}v\text{’} \quad v \neq s \\
icast(w :: ?, \tau) & = icast(w, \tau) & & \\
 & \ldots & & \\
icast(s, \tau) & = icast(number(s), \tau) \quad \tau \in \{\mathbb{Z}, \mathbb{R}\} & & \\
icast(s, \tau) & = s \quad number(s) \text{ is not def.} & &
\end{array}$$

$\boxed{e : \tau \rightsquigarrow\!\!\!\rightsquigarrow \tau'}$ $$\frac{}{e : \tau \rightsquigarrow\!\!\!\rightsquigarrow \tau'}$$

$\boxed{cast(v, \tau) = v'}$ $\ldots$

$$cast(s, \tau) = cast(number(nprefix(s)), \tau) \quad \tau \in \{\mathbb{Z}, \mathbb{R}\}$$

$$cast(s, \tau) = 0 \quad number(nprefix(s)) \text{ is not defined}, \tau \in \{\mathbb{Z}, \mathbb{R}\}$$

$\boxed{bestCandidate(\tau_1, \tau_2, \overline{\tau_3}) = \tau}$

$$\frac{\{m\} = \arg\min_i((\tau 1 - \tau_{1i}) + (\tau_2 - \tau_{2i}))}{bestCandidate(\tau_1, \tau_2, \overline{\tau_{1i} \times \tau_{2i} \to \tau_{3i}}) = \tau_{3m} \times \tau_{4m} \to \tau_{3m}}$$

$$\tau - \tau = 0 \quad \mathbb{Z} - \mathbb{R} = 0 \quad \text{String} - \mathbb{R} = 1 \quad ? - \tau = 1 \quad \_ - \_ = 2$$

$\boxed{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}$

$$\frac{bestCandidate(\tau_1, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}$$

Basic definitions

$$ty(v) = ? \quad dty(w :: ?) = ? \quad dty(r) = \mathbb{R} \quad dty(s) = \text{String}$$

$$clean(T) = T$$

$$insert(e, \tau, 0_C) = e \quad insert\ (e, \tau, +) = e :: \tau$$

$$ty(0_C) = \{? \times ? \to ?, \mathbb{R} \times \mathbb{R} \to ?, \text{String} \times \text{String} \to ?\}$$

$$ty(+) = \{\mathbb{R} \times \mathbb{R} \to ?\}$$

$$apply(+, v_1, v_2) = v_1 + v_2$$

$$apply(0_C, v_1, v_2) = compare(|v_1'|, |v_2'|) :: ?$$

$$\text{where } resolve(v_1, dty(v_1), v_2, dty(v_2), ty(0_C)) = \tau_1 \times \tau_2 \to \tau_3,$$

$$icast(v_1, \tau_1) = v_1', icast(v_2, \tau_2) = v_2'$$

$$|w :: ?| = w \quad |w| = w$$

Figure 9: Sqlite

$\boxed{e : \tau \rightsquigarrow \tau'}$

$$\frac{}{e : \tau \rightsquigarrow \tau'}$$

$\boxed{biconv(e_1, \tau_1, e_2, \tau_2) = \tau}$

$$\frac{\tau_1 \leftrightsquigarrow \tau_2 : \tau}{biconv(e_1, \tau_1, e_2, \tau_2) = \tau}$$

$\boxed{\tau_1 \leftrightsquigarrow \tau_2 : \tau}$

$$\frac{}{\tau \leftrightsquigarrow \tau : \tau} \qquad \frac{}{\text{String} \leftrightsquigarrow \tau : \text{String}}$$

$$\frac{}{\tau \leftrightsquigarrow \text{String} : \text{String}} \qquad \frac{\tau \neq \text{String}}{\mathbb{R} \leftrightsquigarrow \tau : \mathbb{R}} \qquad \frac{\tau \neq \text{String}}{\tau \leftrightsquigarrow \mathbb{R} : \mathbb{R}}$$

$\boxed{cast(e, \tau) = v}$

$$\begin{aligned}
cast(v, ty(v)) &= v \\
cast(v, \text{String}) &= \text{‘}v\text{’} \quad v \neq s \\
cast(n_1.n_2, \mathbb{Z}) &= n_1 \\
cast(n, \mathbb{R}) &= n \\
cast(n, \mathbb{R}) &= n \\
cast(s, \tau) &= icast(number(s), \tau) \quad \tau \in \{\mathbb{Z}, \mathbb{R}\} \\
cast(w :: ?, \tau) &= cast(w, \tau)
\end{aligned}$$

$\boxed{e : \tau \rightsquigarrow\!\!\!\rightsquigarrow \tau'}$

$$\frac{}{e : \tau \rightsquigarrow\!\!\!\rightsquigarrow \tau'}$$

$\boxed{bestCandidate(\tau_1, \tau_2, \overline{\tau_3}) = \tau}$

$$\frac{\{m\} = \arg\min_i((\tau 1 - \tau_{1i}) + (\tau_2 - \tau_{2i}))}{bestCandidate(\tau_1, \tau_2, \overline{\tau_{1i} \times \tau_{2i} \to \tau_{3i}}) = \tau_{3m} \times \tau_{4m} \to \tau_{3m}}$$

$$\tau - \tau = 0 \quad \mathbb{Z} - \mathbb{R} = 0 \quad \text{String} - \mathbb{R} = 1 \quad \_ - \_ = 2$$

$\boxed{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}$

$$\frac{bestCandidate(\tau_1, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \to \tau_6}$$

Basic definitions

$$ty(r) = \mathbb{R} \quad ty(n) = \mathbb{Z} \quad ty(s) = \text{String}$$

$$clean(T) = T$$

$$insert(e, \tau, 0) = e :: \tau \quad 0 \in \{+, <, =\}$$

$$ty(0_C) = \{\mathbb{R} \times \mathbb{R} \to \mathbb{R}, \text{String} \times \text{String} \to \mathbb{R}\}$$

$$ty(+) = \{\mathbb{R} \times \mathbb{R} \to \mathbb{R}\}$$

Figure 10: Mysql

Appendix

A ALL ENGINES

We show the full SQLite (9) the other engines: MySQL (Figure 10), MSSQL (Figure 11) and Oracle (Figure 12).

B DYNAMIC SEMANTICS

In figure 13, we show the semantics where tables are multiple sets. The list of column names of a query Q is obtained using function $\ell(Q)$, and defined inductively as follows:

$$\begin{aligned}
\ell(R) &= \text{list of attributes of base relation } R \\
\ell(Q_1 \times Q_2) &= \ell(Q_1) +\!\!+\ \ell(Q_2) \\
\ell(Q_1\ 0_S\ Q_2) &= \ell(Q_1) \ \text{ where } 0_S \in \{\cup, \cap, -\};\ (*) \\
\ell(\sigma_\theta(Q)) &= \ell(Q) \\
\ell(\pi_\beta(Q)) &= \ell(\beta) \\
\ell(\beta, e \text{ as } N) &= \ell(\beta), \ell(e \text{ as } N) \\
\ell(e \text{ as } N) &= N
\end{aligned}$$

$\boxed{e : \tau \rightsquigarrow \tau'}$

$$\frac{\tau \Rightarrow \tau'}{e : \tau \rightsquigarrow \tau'} \qquad \frac{}{e : \text{String} \rightsquigarrow \tau}$$

$\boxed{icast(e, \tau) = v}$

$$\begin{aligned} icast(‘n’, ty(v)) &= n \\ icast(‘r’, ty(v)) &= r \\ icast('n', \mathbb{R}) &= n \\ icast(v, ty(v)) &= v \\ icast(n, \mathbb{R}) &= n \end{aligned}$$

$\boxed{biconv(e_1, \tau_1, e_2, \tau_2) = \tau}$

$$\frac{e_1 : \tau_1 \rightsquigarrow \tau_2}{biconv(e_1, \tau_1, e_2, \tau_2) = \tau_2} \qquad \frac{e_2 : \tau_2 \rightsquigarrow \tau_1}{biconv(e_1, \tau_1, e_2, \tau_2) = \tau_1}$$

$\boxed{e : \tau \stackrel{\sim}{\rightsquigarrow} \tau'}$

$$\frac{e \neq v}{e : \tau \stackrel{\sim}{\rightsquigarrow} \tau'} \qquad \frac{cast(v, \tau') = v'}{v : \tau \stackrel{\sim}{\rightsquigarrow} \tau'}$$

$\boxed{cast(e, \tau) = v}$

$$\begin{aligned} cast(n_1.n_2, \mathbb{Z}) &= n_1 \\ cast(v, \text{String}) &= ‘v’ \\ cast(v, t) &= icast(v, t) \end{aligned}$$

$\boxed{bestCandidate(\tau_1, \tau_2, \overline{\tau_3}) = \tau}$

$$\frac{\{m\} = \arg\min_i((\tau 1 - \tau_{1i}) + (\tau_2 - \tau_{2i}))}{bestCandidate(\tau_1, \tau_2, \overline{\tau_{1i} \times \tau_{2i} \longrightarrow \tau_{3i}}) = \tau_{3m} \times \tau_{4m} \longrightarrow \tau_{3m}}$$

$$\tau - \tau = 0 \quad \mathbb{Z} - \mathbb{R} = 1 \quad \text{String} - \mathbb{Z} = 1 \quad \text{String} - \mathbb{R} = 1 \quad \_ - \_ = 2$$

$\boxed{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}$

$$\frac{\begin{array}{c} bestCandidate(\tau_1, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6 \\ e1 : \tau_1 \stackrel{\sim}{\rightsquigarrow} \tau_4 \qquad e2 : \tau_2 \stackrel{\sim}{\rightsquigarrow} \tau_5 \end{array}}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}$$

Basic definitions

$$\begin{aligned} &ty(r) = \mathbb{R} \quad ty(n) = \mathbb{Z} \quad ty(s) = \text{String} \\ &clean(T) = T \\ &insert(e, \tau, 0) = e :: \tau \quad 0 \in \{+, <, =\} \\ &ty(0_C) = \{\mathbb{Z} \times \mathbb{Z} \to \mathbb{B}, \mathbb{R} \times \mathbb{R} \to \mathbb{B}, \text{String} \times \text{String} \to \mathbb{B}\} \\ &ty(+) = \{\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \mathbb{R} \times \mathbb{R} \to \mathbb{R}, \text{String} \times \text{String} \to \text{String}\} \end{aligned}$$

Figure 11: Mssql

where "++" denotes list concatenation, and "," list appending. Rule (*) follows standard engine usage of using the left schema of a set expression as output schema.

$\boxed{e : \tau \rightsquigarrow \tau'}$

$$\frac{\tau \Rightarrow \tau'}{e : \tau \rightsquigarrow \tau'}$$

$\boxed{biconv(e_1, \tau_1, e_2, \tau_2) = \tau}$

$$\frac{e_1 : \tau_1 \rightsquigarrow \tau_2}{biconv(e_1, \tau_1, e_2, \tau_2) = \tau_2} \qquad \frac{e_2 : \tau_2 \rightsquigarrow \tau_1}{biconv(e_1, \tau_1, e_2, \tau_2) = \tau_1}$$

$\boxed{cast(e, \tau) = v}$

$$\begin{aligned} cast('n', \mathbb{Z}) &= n \\ cast('r', \mathbb{R}) &= r \\ cast(v, ty(v)) &= v \\ cast('n', \mathbb{R}) &= n \\ cast('n_1.n_2', \mathbb{Z}) &= n_1 \\ cast(n_1.n_2, \mathbb{Z}) &= n_1 \\ cast(n, \mathbb{R}) &= n \\ cast(v, \text{String}) &= ‘v’ \end{aligned}$$

$\boxed{e : \tau \stackrel{\sim}{\rightsquigarrow} \tau'}$

$$\frac{}{e : \tau \stackrel{\sim}{\rightsquigarrow} \tau'}$$

$\boxed{bestCandidate(\tau_1, \tau_2, \overline{\tau_3}) = \tau}$

$$\frac{\{m\} = \arg\min_i((\tau 1 - \tau_{1i}) + (\tau_2 - \tau_{2i}))}{bestCandidate(\tau_1, \tau_2, \overline{\tau_{1i} \times \tau_{2i} \longrightarrow \tau_{3i}}) = \tau_{3m} \times \tau_{4m} \longrightarrow \tau_{3m}}$$

$$\tau - \tau = 0 \quad \mathbb{Z} - \mathbb{R} = 0 \quad \text{String} - \mathbb{R} = 1 \quad \_ - \_ = 2$$

$\boxed{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}$

$$\frac{bestCandidate(\tau_1, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}{resolve(e_1, \tau_1, e_2, \tau_2, \overline{\tau_3}) = \tau_4 \times \tau_5 \longrightarrow \tau_6}$$

Basic definitions

$$\begin{aligned} &ty(r) = \mathbb{R} \quad ty(n) = \mathbb{Z} \quad ty(s) = \text{String} \\ &clean(T) = T \\ &insert(e, \tau, 0) = e :: \tau \quad 0 \in \{+, <, =\} \\ &ty(0_C) = \{\mathbb{R} \times \mathbb{R} \to \mathbb{B}, \text{String} \times \text{String} \to \mathbb{B}\} \\ &ty(+) = \{\mathbb{R} \times \mathbb{R} \to \mathbb{R}\} \end{aligned}$$

Figure 12: Oracle

C ELABORATION

In figure 14, we show the full elaboration rules.

LEMMA C.1. *If $T \vdash e : \tau \rightsquigarrow e'$ then $\Gamma \vdash e' : \tau$.*

PROOF. The proof is derived easily from the definition. □

THEOREM C.2 (ELABORATION PRESERVE TYPE). *If $\Gamma \vdash Q : T \rightsquigarrow Q'$ then $\Gamma \vdash Q' : T$.*

PROOF. The proof follows by Lemma C.3. □

LEMMA C.3. *If $T \vdash e : \tau$ then there exists a unique e' such that $T \vdash e : \tau \rightsquigarrow e'$.*

$$[\![R]\!]_{\mathsf{D}} = \textbf{ok}\ \mathsf{D}\,(R) \tag{RR}$$

$$[\![Q_1 \circ_{\mathsf{S}} Q_2]\!]_{\mathsf{D}} = \text{do}\begin{cases} t_1 \leftarrow [\![Q_1]\!]_{\mathsf{D}} \\ t_2 \leftarrow [\![Q_2]\!]_{\mathsf{D}} \\ \textbf{ok}\ \varepsilon(t_1 \circ_{\mathsf{S}} t_2) \end{cases} \tag{R$\circ_{\mathsf{S}}$}$$

$$[\![\pi_\beta(Q)]\!]_{\mathsf{D}} = \text{do}\begin{cases} t \leftarrow [\![Q]\!]_{\mathsf{D}} \\ \lceil\{\underbrace{[\![\beta]\!]_{\eta^{\bar{\mathsf{r}}}_{\ell(Q)}}, \ldots, [\![\beta]\!]_{\eta^{\bar{\mathsf{r}}}_{\ell(Q)}}}_{\mathsf{k}\ \text{times}} \mid \bar{\mathsf{r}} \in_{\mathsf{k}} t\}\rceil \end{cases} \tag{Rπ}$$

$$[\![\sigma_\theta(Q)]\!]_{\mathsf{D}} = \text{do}\begin{cases} t \leftarrow [\![Q]\!]_{\mathsf{D}} \\ \_ \leftarrow \lceil\{[\![\theta]\!]_{\eta^{\bar{\mathsf{r}}}_{\ell(Q)}} \mid \bar{\mathsf{r}} \in_{\mathsf{k}} t\}\rceil \\ \textbf{ok}\ \{\underbrace{\bar{\mathsf{r}}, \ldots, \bar{\mathsf{r}}}_{\mathsf{k}\ \text{times}} \mid \bar{\mathsf{r}} \in_{\mathsf{k}} t \wedge [\![\theta]\!]_{\eta^{\bar{\mathsf{r}}}_{\ell(Q)}}\} \end{cases} \tag{Rσ}$$

$$[\![e\ \mathsf{as}\ N]\!]_\eta = [\![e]\!]_\eta \tag{Ras}$$

$$[\![\beta, e\ \mathsf{as}\ N]\!]_\eta = \text{do}\begin{cases} \mathsf{r} \leftarrow [\![\beta]\!]_\eta \\ v \leftarrow [\![e]\!]_\eta \\ \textbf{ok}\ \mathsf{r}, v \end{cases} \tag{Rβ}$$

$$[\![N]\!]_\eta = \textbf{ok}\ \eta(N) \tag{RN}$$

$$[\![v]\!]_\eta = \textbf{ok}\ v \tag{Rv}$$

$$[\![v :: \tau]\!]_\eta = \begin{cases} \textbf{ok}\ v' & \mathit{cast}(v, \tau) = v' \\ \textbf{error} & \text{otherwise} \end{cases} \tag{Rv ::}$$

$$[\![e :: \tau]\!]_\eta = \text{do}\begin{cases} v \leftarrow [\![e]\!]_\eta \\ [\![v :: \tau]\!]_\eta \end{cases} \tag{Re ::}$$

$$[\![\theta_1 \circ_{\mathsf{B}} \theta_2]\!]_\eta = \text{do}\begin{cases} b_1 \leftarrow [\![\theta_1]\!]_\eta \\ b_2 \leftarrow [\![\theta_2]\!]_\eta \\ \textbf{ok}\ b_1 \circ_{\mathsf{B}} b_2 \end{cases} \tag{R$\circ_{\mathsf{B}}$}$$

$$[\![\neg\theta]\!]_\eta = \text{do}\begin{cases} v_1 \leftarrow [\![\theta_1]\!]_\eta \\ \textbf{ok}\ \neg b_1 \end{cases} \tag{R$\neg$}$$

$$[\![e_1 \circ e_2]\!]_\eta = \text{do}\begin{cases} v_1 \leftarrow [\![e_1]\!]_\eta \\ v_2 \leftarrow [\![e_2]\!]_\eta \\ \textbf{ok}\ \mathit{apply}(\circ, v_1, v_2) \end{cases} \tag{R$\circ$}$$

$$\circ \in \{+, =, <\}$$

Figure 13: Dynamic Semantics

Proof. We proceed by induction on the typing $T \vdash e : \tau$.

Case (N). Trivial as there is the only matching translation rule, so we choose $e' = N'$.

Case (v). Trivial as there is the only matching translation rule.

Case ($e :: \tau$). From the induction hypotheses, we know that $T \vdash e : \tau \rightsquigarrow e'$ so the result holds.

Case ($\circ_{\mathsf{C}}, \circ_{\mathsf{A}}$). From the induction hypotheses, we know that $T \vdash e_1 : \tau \rightsquigarrow e_1'$ and $T \vdash e_2 : \tau \rightsquigarrow e_2'$ so the result holds.

□

Lemma C.4. *If $T \vdash \theta : \mathbb{B}$ then there exists a unique θ' such that $T \vdash \theta : \mathbb{B} \rightsquigarrow \theta'$.*

Proof. The proof follows by Lemma C.3. □

Lemma C.5. *If $T \vdash \beta : T'$ then there exists a unique β' such that $T \vdash \beta : T' \rightsquigarrow \beta'$.*

Proof. The proof follows by Lemma C.3. □

Theorem C.6. *If $\Gamma \vdash Q : T$ then there exists a unique Q' such that $\Gamma \vdash Q : T \rightsquigarrow Q'$.*

Proof. We proceed by induction on the typing $\Gamma \vdash Q : T$.

Case (R). Trivial as there is the only matching translation rule, so we choose $Q' = Q$.

Case ($\sigma_\theta(Q)$). For the induction hypotheses, we know that $\Gamma \vdash Q : T \rightsquigarrow Q'$ and by Lemma C.4, we know that $T \vdash \theta : \mathbb{B} \rightsquigarrow \theta'$. Thus the result holds.

Case ($\pi_\beta(Q)$). For the induction hypotheses, we know that $\Gamma \vdash Q : T \rightsquigarrow Q'$ and by Lemma C.5, we know that $T \vdash \beta : T' \rightsquigarrow \beta'$. Thus the result holds.

Case ($\circ_{\mathsf{S}}$). The proof follows by the induction hypotheses and Lemma C.3.

Case ($Q_1 \times Q_2$). For the induction hypotheses, we know that $\Gamma \vdash Q_1 : T_1 \rightsquigarrow Q_1'$ and $\Gamma \vdash Q_2 : T_2 \rightsquigarrow Q_2'$. Thus the result holds.

□

D TYPE SOUNDNESS

In the proof we use some different notations but they are equivalent.

Definition D.1 (Well-formed Mapping).

$$\frac{\forall N \in \mathit{dom}(T).\ \vdash \eta(N) : \tau \qquad \mathit{clean}(\tau) = \mathit{clean}(T(N))}{T \vdash \eta}$$

Definition D.2 (Well-formed Database with Schema).

$$\frac{\forall R \in \mathit{dom}(\Gamma).\cdot \vdash \mathsf{D}(R) : T \qquad \mathit{clean}(T) = \mathit{clean}(\Gamma(R))}{\Gamma \vdash \mathsf{D}}$$

$$\frac{\vdash v : \tau}{\vdash v : (N \mapsto \tau)} \qquad \frac{\vdash r : T \qquad \vdash v : \tau}{\vdash r, v : T, (N \mapsto \tau)}$$

$$\frac{\forall r \in t.\ \vdash r : T}{\vdash t : T} \qquad \frac{}{\vdash \cdot : T}$$

Lemma D.3. *$\forall v\ \tau_1\ \tau_2$, if $\vdash v : \tau_1$ then $(\exists v', \mathit{cast}(v, \tau_2) = v') \vee (\mathit{cast}(v, \tau) = \textbf{error})$.*

Proof. The result follows from the definition of conversion of values and the proof is trivial. □

Lemma D.4. *$\forall v\ \tau_1\ \tau_2$, if $\vdash v : \tau_1$ then $(\exists v', \mathit{cast}(v, \tau_2) = v' \wedge \vdash v' : \tau_3 \wedge \mathit{clean}(\tau_3) = \mathit{clean}(\tau_2)) \vee (\mathit{cast}(v, \tau) = \textbf{error})$.*

Proof. The result follows from the definition of conversion of values and the proof is trivial. □

Lemma D.5. *$\forall T\ e\ \tau\ \eta$, if $T \vdash e : \tau$ and $T \vdash \eta$ then $(\exists v, [\![e]\!]_\eta = v) \vee ([\![e]\!]_\eta = \textbf{error})$.*

Proof. The proof follows by induction on the size of e.

$\boxed{T \vdash e : \tau \leadsto e'}$

$$\text{(E::)}\ \frac{\begin{array}{c} T \vdash e : \tau' \leadsto e' \\ e : \tau' \lessapprox \tau \end{array}}{T \vdash e :: \tau : \tau \leadsto e' :: \tau} \qquad \text{(E}v\text{)}\ \frac{T \vdash v : \tau}{T \vdash v : \tau \leadsto v :: \tau} \qquad \text{(E}N\text{)}\ \frac{T(N) = \tau}{T \vdash N : \tau \leadsto N}$$

$$\text{(E}\mathsf{O}_\mathsf{B}\text{)}\ \frac{T \vdash \theta_1 : \mathbb{B} \leadsto \theta_1' \quad T \vdash \theta_2 : \mathbb{B} \leadsto \theta_2' \quad \mathsf{O}_\mathsf{B} \in \{\wedge, \vee\}}{T \vdash \theta_1\ \mathsf{O}_\mathsf{B}\ \theta_2 : \mathbb{B} \leadsto \theta_1'\ \mathsf{O}\ \theta_2'} \qquad \text{(E}\neg\text{)}\ \frac{T \vdash \theta : \mathbb{B} \leadsto \theta'}{T \vdash \neg\theta : \mathbb{B} \leadsto \neg\theta'}$$

$$\text{(E}\mathsf{O}\text{)}\ \frac{\begin{array}{c} T \vdash e_1 : \tau_1 \leadsto e_1' \qquad\qquad T \vdash e_2 : \tau_2 \leadsto e_2' \\ \mathsf{O} \in \{<, =, +\} \quad \mathit{resolve}(e_1, \tau_1, e_2, \tau_2, \mathit{ty}(\mathsf{O})) = \tau_3 \times \tau_4 \longrightarrow \tau_5 \end{array}}{T \vdash e_1\ \mathsf{O}\ e_2 : \tau_4 \leadsto \mathit{insert}(\mathit{insert}(e_1', \tau_3, \mathsf{O})\ \mathsf{O}\ \mathit{insert}(e_2', \tau_4, \mathsf{O}), \tau_5, \mathsf{O})}$$

$\boxed{T \vdash \beta : T' \leadsto \beta'}$

$$\text{(E}\beta\text{)}\ \frac{\overline{T \vdash e : \tau \leadsto e'} \quad \mathit{unique}(\overline{N})}{T \vdash \overline{e\ \mathsf{as}\ N} : \overline{N \mapsto \tau} \leadsto \overline{e'\ \mathsf{as}\ N}}$$

$\boxed{\Gamma \vdash Q : T \leadsto Q'}$

$$\text{(E}R\text{)}\ \frac{\Gamma(R) = T}{\Gamma \vdash R : T \leadsto R} \qquad \text{(E}\pi\text{)}\ \frac{\Gamma \vdash Q : T \leadsto Q' \quad \mathit{clean}(T) \vdash \beta : T' \leadsto \beta'}{\Gamma \vdash \pi_\beta(Q) : T' \leadsto \pi_{\beta'}(Q')} \qquad \text{(E}\sigma\text{)}\ \frac{\Gamma \vdash Q : T \leadsto Q' \quad T \vdash \theta : \mathbb{B} \leadsto \theta'}{\Gamma \vdash \sigma_\theta(Q) : T \leadsto \sigma_{\theta'}(Q')}$$

$$\text{(T}\pi\beta\text{)}\ \frac{\begin{array}{c} \Gamma \vdash \pi_{\beta_1}(Q_1) : T_1 \leadsto \pi_{\beta_1'}(Q_1') \quad \Gamma \vdash \pi_{\beta_2}(Q_2) : T_2 \leadsto \pi_{\beta_2'}(Q_2') \\ \mathit{biconv}(\beta_1, T_1, \beta_2, T_2) = T \qquad\qquad \mathsf{O}_\mathsf{S} \in \{\cup, \cap, -\} \end{array}}{\Gamma \vdash \pi_{\beta_1}(Q_1)\ \mathsf{O}_\mathsf{S}\ \pi_{\beta_2}(Q_2) : T \leadsto \pi_{\mathit{insert}(\beta_1', T, \mathsf{O}_\mathsf{S})}(Q_1')\ \mathsf{O}_\mathsf{S}\ \pi_{\mathit{insert}(\beta_2', T, \mathsf{O}_\mathsf{S})}(Q_2')} \qquad \text{(E}\times\text{)}\ \frac{\begin{array}{c} \ell(Q_1) \cap \ell(Q_2) = \emptyset \\ \Gamma \vdash Q_1 : T_1 \leadsto Q_1' \quad \Gamma \vdash Q_2 : T_2 \leadsto Q_2' \end{array}}{\Gamma \vdash Q_1 \times Q_2 : T_1, T_2 \leadsto Q_1' \times Q_2'}$$

Figure 14: Elaboration rules

Case (N).
we know that

(1) $T \vdash \eta$

(2) $\dfrac{T(N) = \tau}{T \vdash N : \tau}$

we need to show $(\exists v, [\![N]\!]_\eta = v) \vee ([\![N]\!]_\eta = \textbf{error})$
from 1, we know that

$$\frac{\forall N' \in \mathit{dom}(T).\ \vdash \eta(N') : \tau' \quad \mathit{clean}(\tau') = \mathit{clean}(T(N'))}{T \vdash \eta}$$

from 2, we know that $N \in \mathit{dom}(T)$
so
$[\![N]\!]_\eta = \eta(N)$
thus the result holds.

Case (v).
$[\![v]\!]_\eta = v$
thus the result holds.

Case ($e :: \tau$).
we know that

(1) $T \vdash \eta$

(2) $\dfrac{\begin{array}{c} T \vdash e : \tau' \\ e : \tau' \lessapprox \tau \end{array}}{T \vdash e :: \tau : \tau}$

we need to show
$(\exists v, [\![e :: \tau]\!]_\eta = v) \vee ([\![e :: \tau]\!]_\eta = \textbf{error})$
from the hypothesis, we know
$(\exists v, [\![e]\!]_\eta = v) \vee ([\![e]\!]_\eta = \textbf{error})$
from Lemma D.3, we know that
$(\exists v', \mathit{cast}(v, \tau) = v') \vee (\mathit{cast}(v, \tau) = \textbf{error})$
thus the result holds.

Case ($e_1\ \mathsf{O}_\mathsf{A}\ e_2$).
we know that

(1) $T \vdash \eta$

(2) $\dfrac{T \vdash e_1 : \tau_1 \quad T \vdash e_2 : \tau_2}{T \vdash e_1\ \mathsf{O}_\mathsf{A}\ e_2 : \tau_3}$

we need to show
$(\exists v, [\![e_1\ \mathsf{O}_\mathsf{A}\ e_2]\!]_\eta = v) \vee ([\![e_1\ \mathsf{O}_\mathsf{A}\ e_2]\!]_\eta = \textbf{error})$
from the hypothesis, we know
$(\exists v_1, [\![e_1]\!]_\eta = v_1) \vee ([\![e_1]\!]_\eta = \textbf{error})$
$(\exists v_2, [\![e_2]\!]_\eta = v_2) \vee ([\![e_2]\!]_\eta = \textbf{error})$
after the real implement of O_A,
$v_1\ [\![\mathsf{O}_\mathsf{A}]\!]\ v_2 = v$
thus the result holds.

□

Lemma D.6. *$\forall T\ e\ \tau\ \eta$, if $T \vdash e : \tau$ and $T \vdash \eta$ then* $(\exists v, [\![e]\!]_\eta = v \wedge \vdash v : \tau' \wedge \mathit{clean}(\tau') = \mathit{clean}(\tau)) \vee ([\![e]\!]_\eta = \textbf{error})$.

Proof. The proof follows by induction on the size of e.

Case (N).
we know that

(1) $T \vdash \eta$

(2) $\dfrac{T(N) = \tau}{T \vdash N : \tau}$

we need to show $(\exists v, [\![N]\!]_\eta = v \wedge \vdash v : \tau' \wedge clean(\tau') = clean(\tau)) \vee ([\![N]\!]_\eta = \textbf{error})$
from 1, we know that

$$\frac{\forall N' \in dom(T).\ \vdash \eta(N') : \tau' \qquad clean(\tau') = clean(T(N'))}{T \vdash \eta}$$

from 2, we know that $N \in dom(T)$
so
$[\![N]\!]_\eta = \eta(N)$
$\vdash \eta(N) : \tau'$
thus the result holds.

Case (v).
$[\![v]\!]_\eta = v$
thus the result holds.

Case $(e :: \tau)$.
we know that

(1) $T \vdash \eta$

(2) $$\frac{\begin{array}{c} T \vdash e : \tau' \\ e : \tau' \rightsquigarrow \tau \end{array}}{T \vdash e :: \tau : \tau}$$

we need to show
$(\exists v, [\![e :: \tau]\!]_\eta = v \wedge \vdash v : \tau'' \wedge clean(\tau'') = clean(\tau)) \vee ([\![e :: \tau]\!]_\eta = \textbf{error})$
from the hypothesis, we know
$(\exists v, [\![e]\!]_\eta = v \wedge \vdash v : \tau_1 \wedge clean(\tau_1) = clean(\tau')) \vee ([\![e]\!]_\eta = \textbf{error})$
from Lemma D.4, we know that
$(\exists v', cast(v, \tau) = v' \wedge \vdash v' : \tau_2 \wedge clean(\tau_2) = clean(\tau)) \vee (cast(v, \tau) = \textbf{error})$
thus the result holds.

Case $(e_1\ \mathsf{O_A}\ e_2)$.
we know that

(1) $T \vdash \eta$

(2) $$\frac{T \vdash e_1 : \tau_1 \qquad T \vdash e_2 : \tau_2}{T \vdash e_1\ \mathsf{O_A}\ e_2 : \tau_3}$$

we need to show
$(\exists v, [\![e_1\ \mathsf{O_A}\ e_2]\!]_\eta = v \wedge \vdash v : \tau_3' \wedge clean(\tau_3') = clean(\tau_3)) \vee ([\![e_1\ \mathsf{O_A}\ e_2]\!]_\eta = \textbf{error})$
from the hypothesis, we know
$(\exists v_1, [\![e_1]\!]_\eta = v_1 \wedge \vdash v_1 : \tau_1' \wedge clean(\tau_1') = clean(\tau_1)) \vee ([\![e_1]\!]_\eta = \textbf{error})$
$(\exists v_2, [\![e_2]\!]_\eta = v_2 \wedge \vdash v_2 : \tau_2' \wedge clean(\tau_2') = clean(\tau_2)) \vee ([\![e_2]\!]_\eta = \textbf{error})$
after the real implement of $\mathsf{O_A}$,
$v_1\ [\![\mathsf{O_A}]\!]\ v_2 = v \wedge \vdash v : \tau_3' \wedge clean(\tau_3') = clean(\tau_3)$
thus the result holds.

□

Lemma D.7. *$\forall T\ \theta\ \tau\ \eta$, if $T \vdash \theta : \mathbb{B}$ and $T \vdash \eta$ then $(\exists b, [\![\theta]\!]_\eta = b) \vee ([\![\theta]\!]_\eta = \textbf{error})$.*

Proof. The proof follows by Lemma D.6. □

Lemma D.8. *$\forall T\ \beta\ T'\ \eta$, if $T \vdash \beta : T'$ and $T \vdash \eta$ then $(\exists \mathsf{r}, [\![\beta]\!]_\eta = \mathsf{r}) \vee ([\![\beta]\!]_\eta = \textbf{error})$.*

Proof. We induction on the typing.

Case $(\beta = (e\ \mathsf{as}\ N))$. since $[\![\beta]\!]_\eta = [\![e]\!]_\eta$
by Lemma D.6
there are two cases:
if $[\![e]\!]_\eta = \textbf{error}$ then $[\![\beta]\!]_\eta = \textbf{error}$, the result holds.
else:
$\exists v', [\![e]\!]_\eta = v$
thus the result holds.

Case $(\beta = \beta', (e\ \mathsf{as}\ N))$. we know
$[\![\beta]\!]_\eta = [\![\beta']\!]_\eta, [\![e]\!]_\eta$
from the induction hypothesis,
$(\exists\ \mathsf{r}\ T_1', [\![\beta']\!]_\eta = \mathsf{r}) \vee ([\![\beta']\!]_\eta = \textbf{error})$
if $[\![\beta']\!]_\eta = \textbf{error}$ then $[\![\beta]\!]_\eta = \textbf{error}$
the result holds.
else:
by Lemma D.6, we have two cases:
if $[\![e]\!]_\eta = \textbf{error}$ then $[\![\beta]\!]_\eta = \textbf{error}$
thus the result holds
else:
$\exists v\ \tau', [\![e]\!]_\eta = v$
so $[\![\beta]\!]_\eta = [\![\beta']\!]_\eta, [\![e]\!]_\eta = \mathsf{r}, v$
thus the result holds.

□

Lemma D.9. *$\forall T\ \beta\ T'\ \eta$, if $T \vdash \beta : T'$ and $T \vdash \eta$ then $(\exists \mathsf{r}\ T'', [\![\beta]\!]_\eta = \mathsf{r} \wedge \vdash \mathsf{r} : T'' \wedge clean(T'') = clean(T')) \vee ([\![\beta]\!]_\eta = \textbf{error})$*

Proof. We induction on the typing.

Case $(\beta = (e\ \mathsf{as}\ N))$. since $[\![\beta]\!]_\eta = [\![e]\!]_\eta$
by Lemma D.6 and the typing rule,

$$\frac{T' \vdash e : \tau}{T' \vdash (e\ \mathsf{as}\ N) : (N \mapsto \tau)}$$

so there are two cases:
if $[\![e]\!]_\eta = \textbf{error}$ then $[\![\beta]\!]_\eta = \textbf{error}$, the result holds.
else:
$\exists v'\ \tau', [\![e]\!]_\eta = v \wedge \vdash v : \tau' \wedge clean(\tau') = clean(\tau)$
then $\vdash v : N \mapsto \tau'$ and $clean(N \mapsto \tau') = clean(N \mapsto \tau)$
thus the result holds.

Case $(\beta = \beta', (e\ \mathsf{as}\ N))$. we know
$[\![\beta]\!]_\eta = [\![\beta']\!]_\eta, [\![e]\!]_\eta$

$$\frac{T' \vdash \beta' : T_1 \qquad T' \vdash e\ \mathsf{as}\ N : N \mapsto \tau \qquad N \notin dom(T_1)}{T \vdash (\beta', e\ \mathsf{as}\ N) : T_1, N \mapsto \tau}$$

from the induction hypothesis,
$(\exists\ \mathsf{r}\ T_1', [\![\beta']\!]_\eta = \mathsf{r} \wedge \vdash \mathsf{r} : T_1' \wedge clean(T_1') = clean(T_1)) \vee ([\![\beta']\!]_\eta = \textbf{error})$
if $[\![\beta']\!]_\eta = \textbf{error}$ then $[\![\beta]\!]_\eta = \textbf{error}$
the result holds.
else:
by Lemma D.6, we have two cases:
if $[\![e]\!]_\eta = \textbf{error}$ then $[\![\beta]\!]_\eta = \textbf{error}$
thus the result holds
else: $\exists v\ \tau', [\![e]\!]_\eta = v \wedge \vdash v : \tau' \wedge clean(\tau') = clean(\tau)$
so $[\![\beta]\!]_\eta = [\![\beta']\!]_\eta, [\![e]\!]_\eta = \mathsf{r}, v \wedge \vdash \mathsf{r}, v : (T_1', N \mapsto \tau')$
thus the result holds.

□

Lemma D.10. *If* $e_1 : \tau_1 \rightsquigarrow \tau_2$ *and* $e_1 : \tau_2 \rightsquigarrow \tau_1$ *then* $\tau_1 = \tau_2$.

Proof. From $\tau_1 \Rightarrow \tau_2$, we know that if $\tau_1 \Rightarrow \tau_2$ and $\tau_2 \Rightarrow \tau_1$ then $\tau_1 = \tau_2$, so the result holds. □

Theorem D.11 (Type Safety). $\forall T\ Q\ \mathsf{D}$, *if* $\Gamma \vdash Q : T$ *and* $\Gamma \vdash \mathsf{D}$ *then* $(\exists t, \llbracket Q \rrbracket_{\mathsf{D}} = t) \vee (\llbracket Q \rrbracket_{\mathsf{D}} = \textbf{error})$.

Proof. The proof follows by induction on the size of Q.

Case (R).
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\Gamma(R) = T}{\Gamma \vdash R : T}$

we need to show $\exists t, \llbracket R \rrbracket_{\mathsf{D}} = t$
from 1, we know that

$$\frac{\forall R' \in dom(\Gamma). \vdash \mathsf{D}(R') : T'' \qquad clean(T'') = clean(\Gamma(R'))}{\Gamma \vdash \mathsf{D}}$$

from 2, we know that
$\Gamma(R) = T$
so $R \in dom(\Gamma)$
then there is a $\mathsf{D}(R)$
thus the result holds.

Case $(\sigma_\theta(Q))$.
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\Gamma \vdash Q : T \qquad T \vdash \theta : \mathbb{B}}{\Gamma \vdash \sigma_\theta(Q) : T}$

we need to show
$(\exists t\ T', \llbracket \sigma_\theta(Q) \rrbracket_{\mathsf{D}} = t) \vee (\llbracket \sigma_\theta(Q) \rrbracket_{\mathsf{D}} = \textbf{error})$
from the hypothesis, we know that
$(\exists t', \llbracket Q \rrbracket_{\mathsf{D}} = t') \vee (\llbracket Q \rrbracket_{\mathsf{D}} = \textbf{error})$
if $\llbracket Q \rrbracket_{\mathsf{D}} = \textbf{error}$ then $(\llbracket \sigma_\theta(Q) \rrbracket_{\mathsf{D}} = \textbf{error})$
thus the result holds.
else $(\exists t', \llbracket Q \rrbracket_{\mathsf{D}} = t')$,
$\forall \mathsf{r} \in t'$, from Lemma D.7, we know that
$(\exists b, \llbracket \theta \rrbracket_{\eta^{\mathsf{r}}_{\ell(Q)}} = b) \vee (\llbracket \theta \rrbracket_{\eta^{\mathsf{r}}_{\ell(Q)}} = \textbf{error})$
if $(\llbracket \theta \rrbracket_{\eta^{\mathsf{r}}_{\ell(Q)}} = \textbf{error})$ then $(\llbracket \sigma_\theta(Q) \rrbracket_{\mathsf{D}} = \textbf{error})$
thus the result holds
if $\llbracket \theta \rrbracket_{\eta^{\mathsf{r}}_{\ell(Q)}} =$ true then $\mathsf{r} \in \tau'$ is picked,
else $\mathsf{r} \in \tau'$ is not picked
so the resulting table t is subset of t'
thus the result holds.

Case $(\pi_\beta(Q))$.
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\Gamma \vdash Q : T_1 \qquad T_1' = clean(T_1) \qquad T' \vdash \beta : T_1''}{\Gamma \vdash \pi_\beta(Q) : T_1''}$

we need to show
$(\exists t, \llbracket \pi_\beta(Q) \rrbracket_{\mathsf{D}} = t) \vee (\llbracket \pi_\beta(Q) \rrbracket_{\mathsf{D}} = \textbf{error})$
from the hypothesis, we know that
$(\exists t', \llbracket Q \rrbracket_{\mathsf{D}} = t') \vee (\llbracket Q \rrbracket_{\mathsf{D}} = \textbf{error})$
if $\llbracket Q \rrbracket_{\mathsf{D}} = \textbf{error}$ then $\llbracket \pi_\beta(Q) \rrbracket_{\mathsf{D}} = \textbf{error}$
thus the result holds.
else $\llbracket \pi_\beta(Q) \rrbracket_{\mathsf{D}} = \{\llbracket \beta \rrbracket_{\eta^{\mathsf{r}}_{\ell(Q)}} | \mathsf{r} \in \llbracket Q \rrbracket_{\mathsf{D}}\}$
then by Lemma D.8
thus the result holds.

Case $(Q_1 \mathsf{0}\, Q_2; \mathsf{0} \in \{\cap, \cup, -\})$.

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\begin{array}{c}\Gamma \vdash \pi_{e_1 \mathtt{as} N_1}(Q_1)\ \mathsf{0}\ \pi_{e_2 \mathtt{as} N_2}(Q_2) : (N_1 \mapsto \tau) \\ \Gamma \vdash \pi_{\beta_1}(Q_1)\ \mathsf{0}\ \pi_{\beta_2}(Q_2) : T\end{array}}{\Gamma \vdash \pi_{\beta_1, e_1 \mathtt{as} N_1}(Q_1)\ \mathsf{0}\ \pi_{\beta_2, e_2 \mathtt{as} N_2}(Q_2) : T, N_1 \mapsto \tau}$

we need to show
$(\exists t, \llbracket \pi_{\beta_1, e_1 \mathtt{as} N_1}(Q_1)\ \mathsf{0}\ \pi_{\beta_2, e_2 \mathtt{as} N_2}(Q_2) \rrbracket_{\mathsf{D}} = t)$
$\vee (\llbracket \pi_{\beta_1, e_1 \mathtt{as} N_1}(Q_1)\ \mathsf{0}\ \pi_{\beta_2, e_2 \mathtt{as} N_2}(Q_2) \rrbracket_{\mathsf{D}} = \textbf{error})$
from the hypothesis, we know that
$(\exists t_1, \llbracket \pi_{e_1 \mathtt{as} N_1}(Q_1)\ \mathsf{0}\ \pi_{e_2 \mathtt{as} N_2}(Q_2) \rrbracket_{\mathsf{D}} = t_1)$
$\vee (\llbracket \pi_{e_1 \mathtt{as} N_1}(Q_1)\ \mathsf{0}\ \pi_{e_2 \mathtt{as} N_2}(Q_2) \rrbracket_{\mathsf{D}} = \textbf{error})$
$(\exists t_2, \llbracket \pi_{\beta_1}(Q_1)\ \mathsf{0}\ \pi_{\beta_2}(Q_2) \rrbracket_{\mathsf{D}} = t_2) \vee (\llbracket \pi_{\beta_1}(Q_1)\ \mathsf{0}\ \pi_{\beta_2}(Q_2) \rrbracket_{\mathsf{D}} = \textbf{error})$
so
$(\exists t_1, t_2, \llbracket \pi_{\beta_1, e_1 \mathtt{as} N_1}(Q_1)\ \mathsf{0}\ \pi_{\beta_2, e_2 \mathtt{as} N_2}(Q_2) \rrbracket_{\mathsf{D}} = t_1, t_2)$
$\vee (\llbracket \pi_{\beta_1, e_1 \mathtt{as} N_1}(Q_1)\ \mathsf{0}\ \pi_{\beta_2, e_2 \mathtt{as} N_2}(Q_2) \rrbracket_{\mathsf{D}} = \textbf{error})$
thus the result holds.

Case $(Q_1 \times Q_2)$.
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\begin{array}{c}\ell(Q_1) \cap \ell(Q_2) = \Phi \\ \Gamma \vdash Q_1 : T_1 \qquad \Gamma \vdash Q_2 : T_2\end{array}}{\Gamma \vdash Q_1 \times Q_2 : T_1, T_2}$

we need to show
$(\exists t, \llbracket Q_1 \times Q_2 \rrbracket_{\mathsf{D}} = t) \vee (\llbracket Q_1 \times Q_2 \rrbracket_{\mathsf{D}} = \textbf{error})$
from the hypothesis, we know that
$(\exists t_1, \llbracket Q_1 \rrbracket_{\mathsf{D}} = t_1) \vee (\llbracket Q_1 \rrbracket_{\mathsf{D}} = \textbf{error})$
$(\exists t_2, \llbracket Q_2 \rrbracket_{\mathsf{D}} = t_2) \vee (\llbracket Q_2 \rrbracket_{\mathsf{D}} = \textbf{error})$
so
$(\exists t_1, t_2, \llbracket Q_1 \times Q_2 \rrbracket_{\mathsf{D}} = t_1, t_2) \vee (\llbracket Q_1 \times Q_2 \rrbracket_{\mathsf{D}} = \textbf{error})$
thus the result holds.

□

Theorem D.12 (Type Soundness). $\forall T\ Q\ \mathsf{D}$, *if* $\Gamma \vdash Q : T$ *and* $\Gamma \vdash \mathsf{D}$ *then* $(\exists t, \llbracket Q \rrbracket_{\mathsf{D}} = t$ *and* $\vdash t : T' \wedge clean(T') = clean(T)) \vee (\llbracket Q \rrbracket_{\mathsf{D}} = \textbf{error})$.

Proof. The proof follows by induction on the size of Q.

Case (R).
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\Gamma(R) = T}{\Gamma \vdash R : T}$

we need to show $\exists t, \llbracket R \rrbracket_{\mathsf{D}} = t$ and $\cdot \vdash t : T' \wedge clean(T') = clean(T)$
from 1, we know that

$$\frac{\forall R' \in dom(\Gamma). \vdash \mathsf{D}(R') : T'' \qquad clean(T'') = clean(\Gamma(R'))}{\Gamma \vdash \mathsf{D}}$$

from 2, we know that
$\Gamma(R) = T$
so $R \in dom(\Gamma)$
then there is a $\mathsf{D}(R)$ and $\vdash \mathsf{D}(R) : T'$ and $clean(T') = clean(T)$
so $\llbracket R \rrbracket_{\mathsf{D}} = \mathsf{D}(R)$ and $\vdash \mathsf{D}(R) : T'$
thus the result holds.

Case $(\sigma_\theta(Q))$.
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\Gamma \vdash Q : T \qquad T \vdash \theta : \mathbb{B}}{\Gamma \vdash \sigma_\theta(Q) : T}$

we need to show
$(\exists t\ T', [\![\sigma_\theta(Q)]\!]_{\mathsf{D}} = t \text{ and } \vdash t : T' \wedge \mathit{clean}(T') = \mathit{clean}(T)) \vee ([\![\sigma_\theta(Q)]\!]_{\mathsf{D}} = \textbf{error})$
from the hypothesis, we know that
$(\exists t', [\![Q]\!]_{\mathsf{D}} = t' \text{ and } \vdash t' : T'' \text{ and } \mathit{clean}(T'') = \mathit{clean}(T)) \vee ([\![Q]\!]_{\mathsf{D}} = \textbf{error})$
if $[\![Q]\!]_{\mathsf{D}} = \textbf{error}$ then $([\![\sigma_\theta(Q)]\!]_{\mathsf{D}} = \textbf{error})$
thus the result holds.
else $(\exists t', [\![Q]\!]_{\mathsf{D}} = t' \text{ and } \vdash t' : T'')$,
$\forall \mathsf{r} \in t'$, from Lemma D.7, we know that
$(\exists b, [\![\theta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} = b) \vee ([\![\theta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} = \textbf{error})$
if $([\![\theta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} = \textbf{error})$ then $([\![\sigma_\theta(Q)]\!]_{\mathsf{D}} = \textbf{error})$
thus the result holds
if $[\![\theta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} = \text{true}$ then $\mathsf{r} \in \tau'$ is picked,
else $\mathsf{r} \in \tau'$ is not picked
so the resulting table t is subset of t'
and $\vdash t' : T''$
thus the result holds.

Case $(\pi_\beta(Q))$.
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\Gamma \vdash Q : T_1 \qquad T_1' = \mathit{clean}(T_1) \qquad T' \vdash \beta : T_1''}{\Gamma \vdash \pi_\beta(Q) : T_1''}$

we need to show
$(\exists t, [\![\pi_\beta(Q)]\!]_{\mathsf{D}} = t \text{ and } \vdash t : T_2'' \text{ and } \mathit{clean}(T_2'') = \mathit{clean}(T_1'')) \vee ([\![\pi_\beta(Q)]\!]_{\mathsf{D}} = \textbf{error})$
from the hypothesis, we know that
$(\exists t', [\![Q]\!]_{\mathsf{D}} = t' \text{ and } \vdash t' : T_2 \wedge \mathit{clean}(T_2) = \mathit{clean}(T_1)) \vee ([\![Q]\!]_{\mathsf{D}} = \textbf{error})$
if $[\![Q]\!]_{\mathsf{D}} = \textbf{error}$ then $[\![\pi_\beta(Q)]\!]_{\mathsf{D}} = \textbf{error}$
thus the result holds.
else $[\![\pi_\beta(Q)]\!]_{\mathsf{D}} = \{[\![\beta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} | \mathsf{r} \in [\![Q]\!]_{\mathsf{D}}\}$
then by Lemma D.9
so $\vdash \{[\![\beta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} | \mathsf{r} \in [\![Q]\!]_{\mathsf{D}}\} : T_2'' \wedge \mathit{clean}(T_2'') = \mathit{clean}(T_1'')$
thus the result holds.

Case $(Q_1 \mathsf{0}\, Q_2; \mathsf{0} \in \{\cap, \cup, -\})$.

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\begin{array}{c}\Gamma \vdash \pi_{e_1 \mathsf{as} N_1}(Q_1)\, \mathsf{0}\, \pi_{e_2 \mathsf{as} N_2}(Q_2) : (N_1 \mapsto \tau) \\ \Gamma \vdash \pi_{\beta_1}(Q_1)\, \mathsf{0}\, \pi_{\beta_2}(Q_2) : T\end{array}}{\Gamma \vdash \pi_{\beta_1, e_1 \mathsf{as} N_1}(Q_1)\, \mathsf{0}\, \pi_{\beta_2, e_2 \mathsf{as} N_2}(Q_2) : T, N_1 \mapsto \tau}$

we need to show
$(\exists t, [\![\pi_{\beta_1, e_1 \mathsf{as} N_1}(Q_1)\, \mathsf{0}\, \pi_{\beta_2, e_2 \mathsf{as} N_2}(Q_2)]\!]_{\mathsf{D}} = t \text{ and } \vdash t : N_1 \mapsto \tau', T' \wedge \mathit{clean}(N_1 \mapsto \tau', T') = \mathit{clean}((N_1 \mapsto \tau), T)) \vee ([\![\pi_{\beta_1, e_1 \mathsf{as} N_1}(Q_1)\, \mathsf{0}\, \pi_{\beta_2, e_2 \mathsf{as} N_2}(Q_2)]\!]_{\mathsf{D}} = \textbf{error})$
from the hypothesis, we know that
$(\exists t_1, [\![\pi_{e_1 \mathsf{as} N_1}(Q_1)\, \mathsf{0}\, \pi_{e_2 \mathsf{as} N_2}(Q_2)]\!]_{\mathsf{D}} = t_1 \text{ and } \vdash t_1 : (N_1 \mapsto \tau') \wedge \mathit{clean}((N_1 \mapsto \tau')) = \mathit{clean}((N_1 \mapsto \tau))) \vee ([\![\pi_{e_1 \mathsf{as} N_1}(Q_1)\, \mathsf{0}\, \pi_{e_2 \mathsf{as} N_2}(Q_2)]\!]_{\mathsf{D}} = \textbf{error})$
$(\exists t_2, [\![\pi_{\beta_1}(Q_1)\, \mathsf{0}\, \pi_{\beta_2}(Q_2)]\!]_{\mathsf{D}} = t_2 \text{ and } \vdash t_2 : T' \wedge \mathit{clean}(T') = \mathit{clean}(T)) \vee ([\![\pi_{\beta_1}(Q_1)\, \mathsf{0}\, \pi_{\beta_2}(Q_2)]\!]_{\mathsf{D}} = \textbf{error})$
so
$(\exists t_1, t_2, [\![\pi_{\beta_1, e_1 \mathsf{as} N_1}(Q_1)\, \mathsf{0}\, \pi_{\beta_2, e_2 \mathsf{as} N_2}(Q_2)]\!]_{\mathsf{D}} = t_1, t_2 \text{ and } \vdash t_1, t_2 : (N_1 \mapsto \tau'), T') \vee ([\![\pi_{\beta_1, e_1 \mathsf{as} N_1}(Q_1)\, \mathsf{0}\, \pi_{\beta_2, e_2 \mathsf{as} N_2}(Q_2)]\!]_{\mathsf{D}} = \textbf{error})$
thus the result holds.

Case $(Q_1 \times Q_2)$.
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\begin{array}{c}\ell(Q_1) \cap \ell(Q_2) = \Phi \\ \Gamma \vdash Q_1 : T_1 \qquad \Gamma \vdash Q_2 : T_2\end{array}}{\Gamma \vdash Q_1 \times Q_2 : T_1, T_2}$

we need to show
$(\exists t, [\![Q_1 \times Q_2]\!]_{\mathsf{D}} = t \text{ and } \vdash t : T_1', T_2' \wedge \mathit{clean}(T_1, T_2) = \mathit{clean}(T_1', T_2')) \vee ([\![Q_1 \times Q_2]\!]_{\mathsf{D}} = \textbf{error})$
from the hypothesis, we know that
$(\exists t_1, [\![Q_1]\!]_{\mathsf{D}} = t_1 \text{ and } \vdash t_1 : \wedge \mathit{clean}(T_1') = \mathit{clean}(T_1)) \vee ([\![Q_1]\!]_{\mathsf{D}} = \textbf{error})$
$(\exists t_2, [\![Q_2]\!]_{\mathsf{D}} = t_2 \text{ and } \vdash t_2 : T_2' \wedge \mathit{clean}(T_2') = \mathit{clean}(T_2)) \vee ([\![Q_2]\!]_{\mathsf{D}} = \textbf{error})$
so
$(\exists t_1, t_2, [\![Q_1 \times Q_2]\!]_{\mathsf{D}} = t_1, t_2 \text{ and } \vdash t_1, t_2 : T_1', T_2') \vee ([\![Q_1 \times Q_2]\!]_{\mathsf{D}} = \textbf{error})$
thus the result holds.

□

$\boxed{\mathsf{CF}(Q), \mathsf{CF}(\beta), \mathsf{CF}(e), \mathsf{CF}(\theta)}$

$$\frac{}{\mathsf{CF}(R)} \qquad \frac{\mathsf{CF}(\beta) \qquad \mathsf{CF}(Q)}{\mathsf{CF}(\pi_\beta(Q))} \qquad \frac{\mathsf{CF}(\theta) \qquad \mathsf{CF}(Q)}{\mathsf{CF}(\sigma_\theta(Q))}$$

$$\frac{\mathsf{CF}(Q_1) \qquad \mathsf{CF}(Q_2)}{\mathsf{CF}(Q_1\, \mathsf{0}_{\mathsf{S}}\, Q_2)} \qquad \frac{\mathsf{0} \in \{+, <, =\} \qquad \mathsf{CF}(e_1) \qquad \mathsf{CF}(e_2)}{\mathsf{CF}(e_1\, \mathsf{0}\, e_2)}$$

$$\frac{\mathsf{CF}(\theta_1) \qquad \mathsf{CF}(\theta_2)}{\mathsf{CF}(\theta_1\, \mathsf{0}_{\mathsf{B}}\, \theta_2)} \qquad \frac{}{\mathsf{CF}(N)} \qquad \frac{}{\mathsf{CF}(v)}$$

$$\frac{\mathsf{CF}(e)}{\mathsf{CF}(e \text{ as } N)} \qquad \frac{\mathsf{CF}(e) \qquad \mathsf{CF}(\beta)}{\mathsf{CF}(\beta, e \text{ as } N)}$$

Lemma D.13. *If* $\mathsf{CF}(e)$, $T \vdash \eta$ *and* $T \vdash e : \tau$ *then* $[\![e]\!]_\eta \neq \textbf{error}$.

Proof. We prove by induction on the typing.

Case (N).
we know that

(1) $T \vdash \eta$

(2) $\dfrac{T(N) = \tau}{T \vdash N : \tau}$

we need to show $[\![N]\!]_\eta \neq \textbf{error}$
from 1, we know that
$$\frac{\forall N' \in \mathit{dom}(T).\ \vdash \eta(N') : T(N')}{T \vdash \eta}$$
from 2, we know that $N \in \mathit{dom}(T)$
so
$[\![N]\!]_\eta = \eta(N)$
thus the result holds.

Case (v).
$[\![v]\!]_\eta = v$
thus the result holds.

Case ($e_1 \ \mathsf{o_A} \ e_2$).
we know that

(1) $T \vdash \eta$

(2) $\dfrac{T \vdash e_1 : \tau_1 \quad T \vdash e_2 : \tau_2}{T \vdash e_1 \ \mathsf{o_A} \ e_2 : \tau_3}$

we need to show
$[\![e_1 \ \mathsf{o_A} \ e_2]\!]_\eta \neq \textbf{error}$
from the hypothesis, we know
$[\![e_1]\!]_\eta \neq \textbf{error}$
$[\![e_2]\!]_\eta \neq \textbf{error}$
so $[\![e_1 \ \mathsf{o_A} \ e_2]\!]_\eta \neq \textbf{error}$

□

Lemma D.14. *If* $\mathsf{CF}(\theta)$, $T \vdash \eta$ *and* $T \vdash \theta : \mathbb{B}$ *then* $[\![\theta]\!]_\eta \neq \textbf{error}$.

Proof. The proof follows by Lemma D.13. □

Lemma D.15. *If* $\mathsf{CF}(\beta)$, $T \vdash \eta$ *and* $T \vdash \beta : T'$ *then* $[\![\beta]\!]_\eta \neq \textbf{error}$.

Proof. We prove by induction on the typing.

Case (e as N).
$[\![e \text{ as } N]\!]_\eta = [\![e]\!]_\eta$
from Lemma D.13,
so $[\![e]\!]_\eta \neq \textbf{error}$
thus the result holds.

Case (β, e as N).
$[\![\beta, e \text{ as } N]\!]_\eta = [\![\beta]\!]_\eta, [\![e]\!]_\eta$
from induction hypothesis,
$[\![\beta]\!]_\eta \neq \textbf{error}$
from Lemma D.13,
so $[\![e]\!]_\eta \neq \textbf{error}$
thus the result holds.

□

Theorem D.16 (Free of Runtime Error for Target Language). *If* $\mathsf{CF}(Q)$, $\Gamma \vdash \mathsf{D}$ *and* $\Gamma \vdash Q : T$ *then* $[\![Q]\!]_\mathsf{D} \neq \textbf{error}$.

Proof. We prove by induction on the typing.

Case (R).
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\Gamma(R) = T}{\Gamma \vdash R : T}$

we need to show $[\![R]\!]_\mathsf{D} \neq \textbf{error}$
from 1, we know that

$$\frac{\forall R' \in dom(\Gamma). \vdash \mathsf{D}(R') : clean(\Gamma(R'))}{\Gamma \vdash \mathsf{D}}$$

from 2, we know that
$\Gamma(R) = T$
so $R \in dom(\Gamma)$
then there is a $\mathsf{D}(R)$
and $[\![R]\!]_\mathsf{D} = \mathsf{D}(R)$
thus the result holds.

Case ($\sigma_\theta(Q)$).
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\Gamma \vdash Q : T \quad T \vdash \theta : \mathbb{B}}{\Gamma \vdash \sigma_\theta(Q) : T}$

we need to show
$([\![\sigma_\theta(Q)]\!]_\mathsf{D} \neq \textbf{error})$
from the hypothesis, we know that
$([\![Q]\!]_\mathsf{D} \neq \textbf{error})$
By Lemma D.14,
$\textbf{error} \notin \{\textbf{error} | \mathsf{r} \in [\![Q]\!]_\mathsf{D} \wedge [\![\theta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} = \textbf{error}\}$
then $[\![\sigma_\theta(Q)]\!]_\mathsf{D} = \{\mathsf{r} | \mathsf{r} \in [\![Q]\!]_\mathsf{D} \wedge [\![\theta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}}\}$
thus the result holds.

Case ($\pi_\beta(Q)$).
we know that

(1) $\Gamma \vdash \mathsf{D}$

(2) $\dfrac{\Gamma \vdash Q : T_1 \quad T_1' = clean(T_1) \quad T' \vdash \beta : T_1''}{\Gamma \vdash \pi_\beta(Q) : T_1''}$

we need to show
$[\![\pi_\beta(Q)]\!]_\mathsf{D} \neq \textbf{error}$
from the hypothesis, we know that
$[\![Q]\!]_\mathsf{D} \neq \textbf{error}$
By Lemma D.14,
$\textbf{error} \notin \{[\![\beta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} | \mathsf{r} \in [\![Q]\!]_\mathsf{D}\}$
so $[\![\pi_\beta(Q)]\!]_\mathsf{D} = \{[\![\beta]\!]_{\eta^{\mathsf{r}}_{\ell(Q)}} | \mathsf{r} \in [\![Q]\!]_\mathsf{D}\}$
thus the result holds.

Case ($Q_1 \ \mathsf{o} \ Q_2; \mathsf{o} \in \{\cap, \cup, -, \times\}$).
from the induction hypothesis, we know that
$[\![Q_1]\!]_\mathsf{D} \neq \textbf{error}$
$[\![Q_2]\!]_\mathsf{D} \neq \textbf{error}$
so $Q_1 \ \mathsf{o} \ Q_2 = [\![Q_1]\!]_\mathsf{D} \ \mathsf{o} \ [\![Q_2]\!]_\mathsf{D}$
thus the result holds.

□

Lemma D.17. *If* $icast(v, \tau) = v'$ *then* $[\![v :: \tau]\!] \neq \textbf{error}$.

Proof. From the implicit conversion definition, it is the subset of explicit conversion, thus the result holds. □

Lemma D.18. *If* $\vdash v : \tau_1$ *and* $\tau_1 \Rightarrow \tau_2$ *then* $[\![v :: \tau_2]\!] \neq \textbf{error}$.

Proof. From the definition of $\tau_1 \Rightarrow \tau_2$, the result holds. □

Lemma D.19. *If* $resolve(e_1, \tau_1, e_2, \tau_2, ty(\mathsf{o})) = \tau_3 \times \tau_4 \longrightarrow \tau_5$ *then* $[\![e_1 :: \tau_3]\!] \neq \textbf{error}$ *and* $[\![e_2 :: \tau_4]\!] \neq \textbf{error}$.

Proof. for postgres, the proof follows by Lemma D.17 and Lemma D.18. for mysql and sqlite, it holds because casting is defined for all values and types. □

Lemma D.20. *If* $\mathsf{CF}(e)$, $T \vdash \eta$ *and* $T \vdash e : \tau \rightsquigarrow e'$ *then* $[\![e']\!]_\eta \neq \textbf{error}$.

Proof.

Case (N).
we know that

(1) $T \vdash \eta$

(2) $$(\mathrm{E}N)\ \frac{T(N) = \tau}{T \vdash N : \tau \rightsquigarrow N}$$

we need to show $[\![N]\!]_\eta \neq$ **error**
from 1, we know that
$$\frac{\forall N' \in dom(T).\ \vdash \eta(N') : T(N')}{T \vdash \eta}$$
from 2, we know that $N \in dom(T)$
so
$[\![N]\!]_\eta = \eta(N)$
thus the result holds.

Case (v).
$$(\mathrm{E}v)\ \frac{T \vdash v : \tau}{T \vdash v : \tau \rightsquigarrow v :: \tau}$$
the proof follows by Lemma D.18.

Case $(e_1 \ 0\ e_2)$.
we know that

(1) $T \vdash \eta$

(2) $$\frac{T \vdash e_1 : \tau_1 \rightsquigarrow e_1' \qquad T \vdash e_2 : \tau_2 \rightsquigarrow e_2' \qquad 0 \in \{<, =, +\} \qquad resolve(e_1, \tau_1, e_2, \tau_2, ty(0)) = \tau_3 \times \tau_4 \longrightarrow \tau_5}{T \vdash e_1 \ 0\ e_2 : \tau_4 \rightsquigarrow insert(insert(e_1', \tau_3, 0)\ 0\ insert(e_2', \tau_4, 0), \tau_5, 0)}$$

we need to show
$[\![insert(insert(e_1', \tau_3, 0)\ 0\ insert(e_2', \tau_4, 0), \tau_5, 0)]\!]_\eta \neq$ **error**
from the hypothesis, we know
$[\![e_1]\!]_\eta \neq$ **error**
$[\![e_2]\!]_\eta \neq$ **error**
the result follows by Lemma D.19.

□

Lemma D.21. *If* $\mathsf{CF}(\theta)$, $T \vdash \eta$ *and* $T \vdash \theta : \mathbb{B} \rightsquigarrow \theta'$ *then* $[\![\theta']\!]_\eta \neq$ **error**.

Proof. The proof follows by Lemma D.20. □

Lemma D.22. *If* $\mathsf{CF}(\beta)$, $T \vdash \eta$ *and* $T \vdash \beta : T' \rightsquigarrow \beta'$ *then* $[\![\beta']\!]_\eta \neq$ **error**.

Proof. The proof follows by Lemma D.20. □

Theorem D.23 (Free of Runtime Error). *If* $\mathsf{CF}(Q)$, $\Gamma \vdash \mathsf{D}$ *and* $\Gamma \vdash Q : T \rightsquigarrow Q'$ *then* $[\![Q']\!]_\mathsf{D} \neq$ **error**.

Proof. The proof follows by Lemma D.21 and D.22 and the induction hypothesis. □

E TRANSLATION

We show the full translation from SQL to Relational Algebra in Figure 15.

$$
\begin{array}{rcll}
E & ::= & N \mid v \mid E\ 0_\mathsf{A}\ E \mid \mathtt{CAST}(E\ \mathtt{AS}\ \tau) & \text{(expressions)}\\
C & ::= & E\ 0_\mathsf{C}\ E \mid C\ 0_\mathsf{B}\ C & \text{(boolean expressions)}\\
G & ::= & E \mid E\ \mathtt{AS}\ N & \text{(aliased column)}\\
B & ::= & G \mid B, G & \text{(columns)}\\
S & ::= & \mathtt{SELECT}\ B\ \mathtt{FROM}\ F\ \mathtt{WHERE}\ C \mid & \\
 & & \mathtt{SELECT}\ B\ \mathtt{FROM}\ F \mid S\ 0_\mathsf{S}\ S & \text{(select queries)}\\
F & ::= & R \mid R\ N \mid S\ N \mid F, F & \text{(from queries)}
\end{array}
$$

$\boxed{E \rightsquigarrow_e e, C \rightsquigarrow_e \theta}$

$$(\mathrm{Tr}N)\ \frac{}{N \rightsquigarrow_e N} \qquad (\mathrm{Tr}::)\ \frac{E \rightsquigarrow_e e}{\mathtt{CAST}(E\ \mathtt{AS}\ \tau) \rightsquigarrow_e e :: \tau}$$

$$(\mathrm{Tr}v)\ \frac{}{v \rightsquigarrow_e v} \qquad (\mathrm{Tr}0_\mathsf{A})\ \frac{E_1 \rightsquigarrow_e e_1 \qquad E_2 \rightsquigarrow_e e_2}{E_1\ 0_\mathsf{A}\ E_2 \rightsquigarrow_e e_1\ 0_\mathsf{A}\ e_2}$$

$$(\mathrm{Tr}0_\mathsf{C})\ \frac{E_1 \rightsquigarrow_e e_1 \qquad E_2 \rightsquigarrow_e e_2}{E_1\ 0_\mathsf{C}\ E_2 \rightsquigarrow_e e_1\ 0_\mathsf{C}\ e_2}$$

$$(\mathrm{Tr}0_\mathsf{B})\ \frac{C_1 \rightsquigarrow_e \theta_1 \qquad C_2 \rightsquigarrow_e \theta_2}{C_1\ 0_\mathsf{B}\ C_2 \rightsquigarrow_e \theta_1\ 0_\mathsf{B}\ \theta_2}$$

$\boxed{B \rightsquigarrow_b \beta}$

$$(\mathrm{Tr}E)\ \frac{E\ \mathtt{AS}\ alias(E) \rightsquigarrow_b e}{E \rightsquigarrow_b e} \qquad (\mathrm{Tr}Eas)\ \frac{E \rightsquigarrow_e e}{E\ \mathtt{AS}\ N \rightsquigarrow_b e\ \mathsf{as}\ N}$$

$$(\mathrm{Tr}B)\ \frac{B \rightsquigarrow_b \beta \qquad G \rightsquigarrow_b e}{B, G \rightsquigarrow_b \beta, e}$$

$\boxed{F \rightsquigarrow Q}$

$$(\mathrm{Tr}R)\ \frac{}{R \rightsquigarrow R}$$

$$(\mathrm{Tr}SW)\ \frac{B \rightsquigarrow_b \beta \qquad F \rightsquigarrow Q \qquad C \rightsquigarrow_e \theta}{\mathtt{SELECT}\ B\ \mathtt{FROM}\ F\ \mathtt{WHERE}\ C \rightsquigarrow \pi_\beta(\sigma_\theta(Q))}$$

$$(\mathrm{Tr}S)\ \frac{B \rightsquigarrow_b \beta \qquad F \rightsquigarrow Q}{\mathtt{SELECT}\ B\ \mathtt{FROM}\ F \rightsquigarrow \pi_\beta(Q)}$$

$$(\mathrm{Tr}\times)\ \frac{F_1 \rightsquigarrow Q_1 \qquad F_2 \rightsquigarrow Q_2}{F_1, F_2 \rightsquigarrow Q_1 \times Q_2}$$

$$(\mathrm{Tr}Fas)\ \frac{S \rightsquigarrow Q \qquad \ell(Q) = N_1, \ldots}{S\ N \rightsquigarrow \pi_{N_1 \mathsf{as} N.N_1, \ldots}(Q)}$$

Figure 15: SQL to RA translation

Lemma E.1. *Consider SQL expression* E. *Then there exists a unique* e *such that* $E \rightsquigarrow_e e$.

Proof. We proceed by induction on the structure of E.

Case $(E = N)$. Trivial as $N \rightsquigarrow_e N$ is the only matching translation rule, so we choose $e = N$.

Case $(E = v)$. Similar to previous case, $v \rightsquigarrow_e v$ is the only matching translation rule. So we choose $e = v$.

Case $(E = \mathtt{CAST}(E'\ \mathtt{AS}\ \tau))$. By induction hypothesis on E', we know that $E' \rightsquigarrow_e e$ for some unique e. Then by (Tr::), we can conclude that $\mathtt{CAST}(E\ \mathtt{AS}\ \tau) \rightsquigarrow_e e :: \tau$, for unique $e :: \tau$.

Case $(E = E_1\ 0_\mathsf{A}\ E_2)$. By induction hypotheses on E_1 and E_2, we know that $E_1 \rightsquigarrow_e e_1$ and $E_2 \rightsquigarrow_e e_2$, for some unique e_1 and e_2. Then by $(\mathrm{Tr}0_\mathsf{A})$, we can conclude that $E \rightsquigarrow_e e$, for unique $e = e_1\ 0_\mathsf{A}\ e_2$.

□

Lemma E.2. *Consider SQL expression* C. *Then there exists a unique* θ *such that* $C \rightsquigarrow_e \theta$.

Proof. We proceed by induction on the structure of C.

Case ($C = E_1 \odot_C E_2$). By Lemma E.1 on E_1 and E_2, we know that $E_1 \rightsquigarrow_e e_1$ and $E_2 \rightsquigarrow_e e_2$, for some unique e_1 and e_2. Then by (Tr$\odot_C$), we can conclude that $C \rightsquigarrow_e \theta$, for unique $\theta = e_1 \odot_C e_2$.

Case ($C = C_1 \odot_B C_2$). By Lemma E.1 on C_1 and C_2, we know that $C_1 \rightsquigarrow_e \theta_1$ and $C_2 \rightsquigarrow_e \theta_2$, for some unique θ_1 and θ_2. Then by (Tr$\odot_B$), we can conclude that $C \rightsquigarrow_e \theta$, for unique $\theta = \theta_1 \odot_B \theta_2$.

□

Lemma E.3. *Consider SQL expression B. Then there exists a unique β such that $B \rightsquigarrow_e \beta$.*

Proof. We proceed by induction on the structure of B.

Case ($B = E$). By Lemma E.1 on E, we know that $E \rightsquigarrow_e e$, for a unique e. Suppose a deterministic total function $alias(\cdot)$ that returns a string given an expression. Then if $alias(E) = N$, then by (TrEas) we can conclude that E **AS** $N \rightsquigarrow_e e$ as N, for unique e as N. Then by (TrE) we conclude that $E \rightsquigarrow_e \beta$, for unique $\beta = e$ as N.

Case ($B = E$ **AS** N). By Lemma E.1 on E, we know that $E \rightsquigarrow_e e$, for a unique e. Then by (TrEas) we conclude that $E \rightsquigarrow_e \beta$, for unique $\beta = e$ as N.

Case ($B = B', G$). By induction hypothesis on B' and G we know that $B' \rightsquigarrow_e \beta'$ and $G \rightsquigarrow_e e$ for unique β' and e. Therefore by (TrB), $B \rightsquigarrow_e \beta$, for unique $\beta = \beta', e$.

□

Lemma E.4. *Consider query Q, then $\ell(Q) = \overline{N}$, for a unique $\overline{N}$.*

Proof. Straightforward induction on the structure of Q. □

Lemma E.5. *Consider SQL expression F. Then there exists a unique Q such that $F \rightsquigarrow_e Q$.*

Proof. We proceed by induction on the structure of F.

Case ($F = R$). Trivial as $R \rightsquigarrow_e R$.

Case ($F =$ **SELECT** B **FROM** F' **WHERE** C). By Lemma E.3, $B \rightsquigarrow_e \beta$ for unique β. By induction hypothesis on F', $F' \rightsquigarrow_e Q'$ for unique Q'. By Lemma E.2, $C \rightsquigarrow_e \theta$ for unique θ. We conclude by (TrSW) that $F \rightsquigarrow_e Q$, for unique $Q = \pi_\beta(\sigma_\theta(Q'))$.

Case ($F =$ **SELECT** B **FROM** F'). By Lemma E.3, $B \rightsquigarrow_e \beta$ for unique β. By induction hypothesis on F', $F' \rightsquigarrow_e Q'$ for unique Q'. We conclude by (TrS) that $F \rightsquigarrow_e Q$, for unique $Q = \pi_\beta(Q')$.

Case ($F = F_1, F_2$). By induction hypotheses on F_1 and F_2, $F_1 \rightsquigarrow_e Q_1$ and $F_2 \rightsquigarrow_e Q_2$ for unique Q_1 and Q_2. We conclude by (Tr$\times$) that $F \rightsquigarrow_e Q$, for unique $Q = Q_1 \times Q_2$.

Case ($F = A\ N$). By induction on A we know that $A \rightsquigarrow_e Q'$ for a unique Q'. Also, by Lemma E.4, $\ell(Q') = N_1, \ldots$ for unique $N_1, \ldots$. Therefore by (TrFas), $F \rightsquigarrow_e Q$, for unique $Q = \pi_{N_1 \text{as} N.N_1,\ldots}(Q')$.

□

Theorem E.6. *Consider an SQL query denoted as F, which takes the form* **SELECT** *B* **FROM** *F* **WHERE** *C or* **SELECT** *B* **FROM** *F. Then, there exists a unique relational algebra query, denoted as Q, such that $F \rightsquigarrow_e Q$.*

Proof. Direct consequence of Lemma E.5. □

F VALIDATION

The termination-sensitive validation method ensures a more robust outcome. It involves confirming that when a query is evaluated in language PyRAFT and yields a table, the same query evaluated in the engine produces an identical table. Additionally, if a query in PyRAFT encounters an error, the engine also reports an error.

To achieve this validation we needed to simplify several aspects of the generation of SQL queries, due to query optimizations, which we describe next.

Avoiding subqueries in FROM position.. Consider the PSQL query `SELECT S.c FROM (SELECT CAST(Name AS INT) AS c FROM P) AS S`. This query results in an error as the values in the column `Name` cannot be cast to a number. However, a slight change to the query by evaluating `SELECT 1 FROM (SELECT CAST(Name AS INT) AS c FROM P) AS S` allows the query to run without errors. This change in behavior occurs due to an optimization process, where expressions within subqueries are not evaluated if the selection columns do not depend on the subquery's columns. Therefore, we avoid generating subqueries in the FROM position.

Avoiding conjunctions.. Consider the PSQL query `SELECT 1 FROM P WHERE 5 = CAST('hello' AS FLOAT) AND 2 < 1`. This query fails at runtime because the cast from a non-numerical string to an integer is not defined. Now, let's examine the query `SELECT 1 FROM P WHERE 5 = CAST(Name AS FLOAT) AND 2 < 1`. Here, it does not fail; the condition `2<1` is evaluated before the cast from string to integer. Thanks to the *short-circuit* behavior, the entire condition is deemed false, resulting in an empty result. Given the unpredictable nature of the evaluation order of logical conjunctions in various queries across different database engines, we opt to avoid using conjunctions in the query generator.

Single column selections.. Consider the following MSSQL query

```
(SELECT 1, Age FROM P) INTERSECT
(SELECT 1, Name FROM P)
```

This query fails because string values in column `Name` cannot be cast to an integer. However, a slightly different query does not fail:

```
(SELECT 1, Age FROM A) INTERSECT
(SELECT 2, Name FROM A)
```

In this scenario, MSSQL predicts that the intersection is empty by examining only the first column of both tables. It avoids equating the remaining columns, thereby avoiding the evaluation of the doomed-to-fail cast. To circumvent this optimization issue, our approach avoids generating of selections involving more than one column.

Tailoring for MSSQL. Consider the following MSSQL query

```
(SELECT Age FROM P WHERE 1 < CAST(Name AS INT))
INTERSECT (SELECT Age FROM P)
```

This query results in an error due to the condition in the first subquery. However, the following query:

```
(SELECT Age FROM P WHERE 1 < CAST(Name AS INT))
INTERSECT (SELECT Age+1 FROM P)
```

Yields an empty result because the engine recognizes that the intersection between columns `Age` and `Age+1` is empty before evaluating the condition. To avoid this kind of optimization issues, we avoid generating queries with conditionals.

G EXAMPLES

To illustrate the comparison behavior in SQLite, we provide the following examples:

```
SELECT '0' < 1 FROM R;
SELECT '0' < CAST(1 AS INT) FROM R;
SELECT '0hi' < CAST(1 AS INT) FROM R;
```

The first example evaluates to `0`. Since the (Ev) elaboration rule inserts an explicit cast on every value, both values are cast to ?. Consequently, the chosen candidate is $? \times ? \longrightarrow ?$, and the implicit cast leaves them untouched. Finally, as the dynamic type of both operands, with annotations removed, is String and $\mathbb{R}$ respectively, the comparison function yields `1` as result.

The second example evaluates to `1`. In the process of elaboration, the left expression is cast to the unknown type, while the right expression is cast to an integer type. During the actual evaluation, the most suitable candidate is determined to be $\mathbb{R} \times \mathbb{R} \longrightarrow ?$, which implies an implicit cast of both values into numerical values `0` and `1`, respectively. As both casted values share the same dynamic type, a standard comparison is carried out, resulting in `1`.

The third example evaluates to `0`. It follows a pattern similar to the second example, but with the implicit cast to a numeric type failing on the left expression, resulting in the same string as the output. Consequently, since the dynamic types of both operands differ (string and real), a type comparison is carried out, leading to the final result of `0`.

To illustrate the insert function, let us consider three examples in SQLite:

- Expression `'0'` $< 1 :: \mathbb{Z}$ is elaborated to `'0'` $:: ? < (1 :: ?) :: \mathbb{Z}$, which evaluates to 1 (because one of the operand's type is not unknown, see Section 5.2).
- Expression `'0hi'` $< 1 :: \mathbb{Z}$ is elaborated to `'0hi'` $:: ? < (1 :: ?) :: \mathbb{Z}$, which evaluates to 0. If an explicit cast to int is inserted in both operands then the result would be 1 (to denote true), as $[\![$`'0hi'` $:: \mathbb{Z}]\!]_{\eta} = 0$.
- Expression `'1'`$+1$ is elaborated to $($`'1'` $:: ?) :: \mathbb{Z} + (1 :: ?) :: \mathbb{Z}$, which evaluates to 2.