Language-integrated provenance by trace analysis

Language-integrated provenance by trace analysis

Stefan Fehrenbach
University of Edinburgh
United Kingdom
stefan.fehrenbach@gmail.com

James Cheney
University of Edinburgh and The Alan Turing Institute
United Kingdom
jcheney@inf.ed.ac.uk

Abstract

Language-integrated provenance builds on language-integrated query techniques to make provenance information explaining query results readily available to programmers. In previous work we have explored language-integrated approaches to provenance in Links and Haskell. However, implementing a new form of provenance in a language-integrated way is still a major challenge. We propose a self-tracing transformation and trace analysis features that, together with existing techniques for type-directed generic programming, make it possible to define different forms of provenance as user code. We present our design as an extension to a core language for Links called Links^T, give examples showing its capabilities, and outline its metatheory and key correctness properties.

CCS Concepts • **Information systems → Data provenance**; • **Software and its engineering → Functional languages**;

Keywords language-integrated provenance, language-integrated query, query normalization, provenance

ACM Reference Format:
Stefan Fehrenbach and James Cheney. 2019. Language-integrated provenance by trace analysis. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages (DBPL '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3315507.3330198

1 Introduction

Provenance tracking has been heavily investigated as a means of making database query results explainable [4, 8], for example to explain where in the input some output data came from (*where-provenance*) or what input records justify the presence of some output record (*lineage*, *why-provenance*).

DBPL '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6718-9/19/06…$15.00
https://doi.org/10.1145/3315507.3330198

Many prototype implementations of provenance-tracking have been developed as ad hoc extensions to (or middleware layers wrapping) ordinary relational database systems [3, 25], typically by augmenting the data model with additional annotations and propagating them through the query using an enriched semantics. This approach, however, inhibits reuse and uptake of these techniques since a special (and usually not maintained) variant of the database system must be used. Installing, maintaining and using such research prototypes is not for the faint of heart.

We advocate a *language-based* approach to provenance, building on *language-integrated query* [9, 18, 20]. In language-integrated query, database queries are embedded in a programming language as first-class citizens, not uninterpreted strings, and thus benefit from typechecking and other language services. In language-integrated *provenance*, we aim to support provenance-tracking techniques by modifying the behavior of queries at the language level to track their own provenance. These modified queries can then be used with unmodified, mainstream database systems. To date, Fehrenbach and Cheney [14] have demonstrated the capabilities of language-integrated provenance in Links, a Web and database programming language, and Stolarek and Cheney [26] adapted this approach to work with Dsh, an existing language-integrated query library in Haskell [28]. In both cases, where-provenance and lineage are supported as representative forms of provenance.

However, both approaches explored so far have drawbacks. Our previous implementations of language-integrated provenance in Links are ad hoc language extensions, requiring nontrivial changes to the Links front-end and runtime. It is not obvious how to support both extensions at once, and supporting additional extensions would likewise require a major intervention to the language. In Dsh, we were able to support both forms of provenance at once, but did need to make superficial changes to Dsh and carry out nontrivial type-level programming to make our translations pass Haskell's typechecker. Thus, in both cases, we feel there is significant room for improvement, to make it easier to develop new forms of provenance without ad hoc language extensions or subtle type-level programming.

In this paper, we present a core language design called Links^T that extends the query language core of Links (a variant of the Nested Relational Calculus [5]) with several powerful programming constructs. These include well-studied constructs for type-directed generic programming (e.g. **Typerec**

and **typecase**) [16], extended to support generic programming with record types [10]. In addition, we propose novel primitives for constructing and analyzing *query traces* (following [7]). We will show that these features suffice to define forms of provenance programmatically, using the following recipe. Given a query q, we first *transform* it to a *self-tracing* query q^{T}. We can then *compose* q^{T} with a *trace analysis function* f^{P}, which is simply an ordinary Links$^{\mathsf{T}}$ function that makes use of the type and trace analysis capabilities. Each form of provenance we support can be defined as a trace analysis function, and can be applied to queries of any type. Thus, $f^{\mathsf{P}} \circ q^{\mathsf{T}}$ defines the intended query result together with the desired provenance. Finally, we *normalize* $f^{\mathsf{P}} \circ q^{\mathsf{T}}$ to a Nrc expression, which can be further translated to Sql and evaluated efficiently on a mainstream database by the existing language-integrated query implementation in Links [9]. Normalization effectively deforests the traces that would be produced by q^{T} if we were to execute it directly; thus, executing the normalized Nrc query is typically much faster than executing q^{T} and then f^{P} separately would be.

Our main contributions are as follows.

- We show via examples (Section 4) how a programmer can use type and trace analysis constructs to define different modifications of query behavior, for example to extract where-provenance and lineage from traces.
- We present the language design of Links$^{\mathsf{T}}$. We informally introduce the novel trace constructors in Section 3 and present syntax and type system details in Section 5. This includes traces and trace analysis operations, and reviews the already-studied type-directed generic programming features from previous work.
- We then present the self-tracing transformation (Section 6) and the extended rewrite rule system needed for normalization, and outline the proofs of type preservation and correctness for these components (Section 7).

We have a preliminary implementation, but the main contributions of this paper concern the design and theory, and a full-scale implementation in Links is future work.

2 The problem

As explained earlier, in previous work we have investigated different ways of implementing where-provenance and lineage on top of existing language-integrated query systems, namely Links and Dsh. In both cases, given a query q, we wish to construct another query q^{P} that provides both the ordinary query results of q and additional *annotations* that provide some form of information about how query results relate to the input data. Preferably, the transformed query should still be in the same query language as that handled by the existing language-integrated query system, so that this implementation can be reused to generate efficient Sql queries. Of course, in a typed programming language, we also expect the generated query to be well-typed.

Agencies

(oid)	name	based_in	phone
1	EdinTours	Edinburgh	412 1200
2	Burns's	Glasgow	607 3000

ExternalTours

(oid)	name	destination	type	price (in £)
3	EdinTours	Edinburgh	bus	20
4	EdinTours	Loch Ness	bus	50
5	EdinTours	Loch Ness	boat	200
6	EdinTours	Firth of Forth	boat	50
7	Burns's	Islay	boat	100
8	Burns's	Mallaig	train	40

BoatToursQueryResult

name	phone
EdinTours	412 1200
EdinTours	412 1200
Burns's	607 3000

Figure 1. Example database and boat tours query result.

For example, for where-provenance, we wish to construct query q^{where} in which each data field in the query result is annotated with a *source location* in the input database, which we typically implement as a tuple (R, A, i) consisting of a relation name R, attribute name A, and row identifier (or primary key value) i. Likewise, for lineage, we wish to construct a query q^{lineage} in which each output record is annotated with a collection of references (R, i) to input records that help "witness" or "justify" the presence of the output record.

As a running example, consider the following boat tours query (in Links syntax). It uses nested **for** comprehensions to iterate over two tables, filtering by type and joining on the name columns. It returns a list of records (pairs of field name and value separated by commas and enclosed in angle brackets) containing the agencies names and phone numbers. See Figure 1 for an example input database and result.

```
for (e <- externalTours) where (e.type == "boat")
  for (a <- agencies) where (a.name == e.name)
    [⟨name = e.name, phone = a.phone⟩]
```

The where-provenance translation of this query should annotate the field value Burns's in the result with where-provenance annotation (ExternalTours, name, 7), and the lineage translation should annotate the row (Burns's, 607 3000) with lineage annotation [(Agencies,2), (ExternalTours,7)]. (Note that in lineage, the annotation of each row is a *collection* of input row references; both Links and Dsh can already handle such nested query results [9, 28].)

In our previous work, we have implemented these translations either by directly changing the language implementation (in Links), or by making nontrivial modifications to

a language-integrated query library (in Dsh). While this work shows that it is possible to provide (reasonably efficient) language-integrated provenance via source-to-source translation of queries, both approaches are still nontrivial interventions to an existing implementation, and so developing new forms of provenance, or variations on existing ones, is still a considerable challenge.

If we wish to provide the necessary query transformation capability using high-level programming constructs, then we face two significant challenges. First, transforming the query expression in the direct approaches considered so far relies on fairly heavyweight metaprogramming capabilities, and type-safe metaprogramming by reflection over object languages with binding constructs (such as comprehensions in queries) is a significant challenge. Based on prior work on general forms of provenance such as *traces* [1, 7] or *provenance polynomials* [15], we might hope to avoid the need for heavyweight metaprogramming by computing a single, general form of *query trace* once and for all, and specializing it to different forms of provenance later. However, this raises the question of how to design a suitable tracing framework and how to provide appropriate language constructs that can specialize traces to different forms of provenance, in a type-safe and efficient way. (In particular, we cannot simply reuse the provenance polynomials/semirings framework since it is not able to capture where-provenance [8].)

Second, and related to the previous point, we need to change not only the query *behavior* but also the query *result type*. Specifically, in the type of q^{where}, each field is replaced with a record consisting of the ordinary data value and its where-provenance annotation, whereas in q^{lineage}, each element of a collection in the query result type becomes a pair consisting of the original data and a *collection* of input row references. In previous implementations, we have added this behavior to the typechecker directly (in Links), or (in Dsh) used *type families* [6] to define the effect of the where-provenance or lineage transformations at the type level. In the case of Dsh, this necessitated subtle changes to the Dsh library, as well as defining evidence translations at the type and term levels to convince Haskell's typechecker that our definitions were type-correct.

Thus, in both Links and Dsh, our previous work has shown that it is possible to implement language-integrated provenance, but the need to manipulate both query expressions and their types makes this more difficult than we might hope. Our goal, therefore, is to identify a small set of language features that addresses all of the above needs well: we would like to be able to customize the query behavior to handle multiple forms of provenance, while retaining the existing benefits demonstrated by previous implementations of language-integrated provenance: specifically type-safety and efficient query generation.

```
TRACE = λa.Typerec a (Trace Bool, Trace Int, Trace String,
                      λe e'.[e'], λr r'.⟨r'⟩, λb t.t)
```

Figure 2. The type-level function TRACE.

3 Query traces

In this section we describe what our traces look like through a series of examples. We show how to rewrite expressions to compute their own trace in Section 6. As described earlier, the intent is to compose a trace analysis function with a self-tracing query and normalize to deforest the trace and only compute the parts that we actually need.

The **trace** keyword causes a query expression to be traced. For example, `trace 2+3` has type `Trace Int` and evaluates to `OpPlus⟨l=Lit 2,r=Lit 3⟩`. Here, `OpPlus` represents an addition operation and its argument is a record of the left and right subtraces, and `Lit` is the constructor for traces of literal values. Traces of records are just records of traces, and traces of lists are just lists of traces, e.g., tracing the singleton list of the singleton record `[⟨answer=42⟩]` results in `[⟨answer=Lit 42⟩]`.

In general, the trace of an expression with type A has a type where every base type is replaced by the traced version of the base type, but all list and record constructors stay the same. We can express this in Links$^{\text{T}}$ directly as the type-level function TRACE defined in Figure 2. We capitalize type-level entities (except variables) and trace constructors, and write type-level functions in all uppercase. **Typerec** folds over a type, in this case the type variable `a`. It uses its first three arguments for base types (in our case replacing `Bool`with `Trace Bool`, etc.). The next argument is used if the argument is a list type and applied to the original element type and the recursively transformed element type. The next arguments work similarly for records and trace types.

Tables are typed as lists of records. Their traces reveal that they are not constants in the query however. Values originating from tables are marked with the `Cell` constructor. For example, the trace of the agencies table looks like this:

```
[⟨oid=Cell⟨tbl="agencies",col="oid",row=1,val=1⟩,
  name=Cell⟨tbl="agencies",col="name",row=1,val="EdinTours"⟩,
  based_in=Cell⟨tbl="agencies",col="based_in",row=1,val="Edinburgh"⟩
  phone=Cell⟨tbl="agencies",col="phone",row=1,val="412 1200"⟩⟩,
 ⟨oid=Cell⟨tbl="agencies",col="oid",row=2,val=2⟩,
  name=Cell⟨tbl="agencies",col="name",row=2,val="Burns's"⟩,
  based_in=Cell⟨tbl="agencies",col="based_in",row=2,val="Glasgow"⟩
  phone=Cell⟨tbl="agencies",col="phone",row=2,val="607 3000"⟩⟩]
```

Conditional expressions record the trace of the condition as well as the trace of the eventually produced result. Polymorphic operations such as `==` record the type they were applied to. The trace of a **for** comprehension carries both the element type of the input collection and subtraces of both the input and the output. For example, the following query is a convoluted way to get `["Edinburgh"]`.

```
for (a <- table "agencies" ...) where (a.name == "EdinTours")
  [a.based_in]
```

Its trace is shown below. We treat **where** (M) N as syntactic sugar for **if** M **then** N **else** [].

```
[ If⟨cond=OpEq String
       ⟨l=For ⟨oid:Int,name:String,...⟩
            ⟨in=⟨oid=Cell⟨tbl="agencies",col="oid",...⟩,
                 name=Cell⟨tbl="agencies",col="name",...⟩,
                 based_in=Cell⟨tbl="agencies",...⟩
                 phone=Cell⟨tbl="agencies",col="phone",...⟩⟩,
             out=Cell⟨tbl="agencies",...⟩,
        r=Lit "EdinTours"⟩,
     out=For ⟨oid:Int,name:String,based_in:String,phone:String⟩
       ⟨in=⟨oid=Cell⟨tbl="agencies",col="oid",...⟩,
            name=Cell⟨tbl="agencies",col="name",...⟩,
            based_in=Cell⟨tbl="agencies",...⟩
            phone=Cell⟨tbl="agencies",col="phone",...⟩⟩,
        out=Cell⟨tbl="agencies",col="based_in",row=1,
                 val="Edinburgh"⟩⟩⟩⟩ ]
```

Note that the variable a does not appear explicitly in the trace. Rather, wherever a variable in an expression would produce a value, we record the subtrace of the value in the trace. Also note that the trace of this singleton list is still a singleton list, and the comprehension marker appears on the (singleton) element. This is a significant deviation from previous work on tracing queries [7] which will make trace analysis much easier as trace analysis functions will not have to deal with variable binding.

4 Trace analysis

Trace analysis functions need to be flexible enough to work with queries of any type and any shape. The shape of a query, and thus the depth of its trace, are not even necessarily known until runtime of the program. Therefore trace analysis functions need to be polymorphic and recursive. In the following we use Λ for term-level type abstraction, **fix** to define recursive values, **typecase** to branch on types, and **tracecase** to branch on trace constructors. We will also use generic record operations to work with records of any number and type of fields. We will describe these in more detail in Section 5.

4.1 Where-provenance

Where-provenance annotates every cell of a query result with information about where in the database the value was copied from. Figure 3 shows the `wherep` trace analysis function and helpers. On the type level, `WHERE` replaces every base type by a record with fields for the value, table, column, and row number. For any type a, `wherep` takes a trace and returns a where-provenance–annotated value. `T()` wraps type-level computation, as explained later. To recover where-provenance from a trace, `wherep` distinguishes three cases: did the traced expression have a list type, a record type, or a

```
W = λa:Type.⟨val:a, tbl:String, col:String, row:Int⟩

WHERE = λa:Type.Typerec a (W Bool, W Int, W String,
                  λ_ b.List b, λ_ r.Record r, λ_ b.b)

wherep : ∀a.T(TRACE a) -> T(WHERE a)
wherep = fix (wherep:∀a.T(TRACE a) -> T(WHERE a)).Λa:Type.
  typecase a of
    List b   => λxs.for (x <- xs) [wherep b x]
    Record r => λx.rmap^r wherep x
    Trace b  => λx.tracecase x of
      Lit y    => fake b y
      If y     => wherep (Trace b) y.out
      For c y  => wherep (Trace b) y.out
      Cell y   => y
      OpPlus y => fake Int (value (Trace Int) x)
      OpEq c y => fake Bool (value (Trace c) x)

fake : ∀a.T(a) -> T(W a)
fake = Λa.λx:T(a).⟨val=x,tbl="facts",col="alternative",row=-1⟩
```

Figure 3. The `wherep` trace analysis function and supporting definitions.

base type. In case of a list type, we map `wherep` over the list of subtraces. (We use a comprehension here, but Links handles higher-order functions like `map` and `filter` just fine.) In case of a record type, we use **rmap** to map `wherep` over the fields of the record of subtraces. In case the original expression was of some base type A, the trace has type `Trace` A, which we further analyze using **tracecase**. If the trace constructor is `Lit` the value was a constant in the query and we need to mark it with fake provenance. In the `If` and `For` cases, we continue extracting where-provenance from their output. If the trace constructor is `Cell`, the value originated from the database and already carries the table and column names and row number. Finally, we associate fake where-provenance with the results of operators, whose value is computed by the `value` trace analysis function (see Section 4.2).

4.2 Value

The `value` trace analysis function is the inverse to tracing. It recovers a plain value from a trace by recomputing values from operators' subtraces and otherwise throwing away all tracing information. It is defined in Appendix A.

4.3 Lineage

This implementation of lineage aims to emulate the behavior of Links$^{\mathrm{L}}$, a variant of Links with built-in support for lineage [14]. This is complicated by the fact that lineage annotations in Links$^{\mathrm{L}}$ are on rows (or more generally, list elements) but tracing information in Links$^{\mathrm{T}}$ is on cells. We need to collect annotations from the trace leaves and pull them up to the nearest enclosing list constructor.

```
L = λa:Type.⟨data: a, lineage: [⟨table: String, row: Int⟩⟩

LINEAGE = λa:Type.Typerec a (Bool, Int, String,
                  λ_ b.List (L b), λ_ r.Record r, λ_ b.b)

lineage : ∀a.T(TRACE a) -> T(LINEAGE a)
lineage = fix (lineage:∀a.T(TRACE a) -> T(LINEAGE a)).Λa:Type.
  typecase a of
    List b   => λts.for (t <- ts)
                  [⟨data = lineage b t,
                    lineage = linnotation b t⟩]
    Record r => λx.rmap^r lineage x
    Trace b  => λx.value (Trace b) x

linnotation : ∀a.T(TRACE a) -> [⟨table: String, row: Int⟩]
linnotation = fix (linnotation: ...).Λa:Type.
  typecase a of
    List b   => λts.for (t <- ts) linnotation b t
    Record r => λx.rfold^Rmap (λ_.[⟨table:String, row:Int⟩]) r (++) []
                  (rmap^r linnotation x)
    Trace b  => λt.tracecase t of
      Lit c    => []
      If i     => linnotation (TRACE b) i.out
      For c f  => linnotation (TRACE c) f.in ++
                  linnotation (TRACE b) f.out
      Cell r   => [⟨table = r.table, row = r.row⟩]
      OpEq c e => linnotation (TRACE c) e.left ++
                  linnotation (TRACE c) e.right
      OpPlus p => linnotation (TRACE Int) p.left ++
                  linnotation (TRACE Int) p.right
```

Figure 4. The `lineage` trace analysis function and supporting definitions.

The `LINEAGE` type function changes list types to carry a list of annotations. On the value level, the implementation is split into two functions: `lineage` and `linnotation`, as shown in Figure 4. The `lineage` function matches on the type of its argument and makes (recursive) calls to `lineage`, `linnotation`, and `value` as appropriate to combine annotations and values. The `linnotation` function does the actual work of computing lineage annotations from traces. The case for lists concatenates the lineage annotations obtained by calling `linnotation` on the list elements. In the case for records, we first use **`rmap`** to map `linnotation` over the record, then we use **`rfold`** to flatten the record of lists of lineage annotations into a single list. Trace constructors have lineage annotations as follows. Literals do not have lineage. Conditional expressions have the lineage of their result. Comprehensions are the interesting case, where we combine lineage annotations from the input with lineage annotations from the output. Each table cell has the expected initial singleton annotation consisting of its table's name and its row number. Finally, the operators just collect their arguments' annotations.

There is an issue with this implementation of lineage: we collect duplicate annotations. Consider the following query:

```
for (x <- table "xs" ⟨a: Int, b: Bool⟩) [x.a]
```

We just project a table to one of its columns. The lineage of every element of the result should be one of the rows in the table. If we apply the `lineage` trace analysis function to the trace of the above query (at the appropriate type) and normalize, we get this query expression:

```
for (x <- table "xs" ⟨a: Int, b: Bool⟩)
  [⟨data=x.a, lineage=[⟨tbl="xs",row=x.oid⟩] ++
           [⟨tbl="xs",row=x.oid⟩] ++ [⟨tbl="xs",row=x.oid⟩]⟩]
```

The lineage is correct, but there is too much of it. Instead of having one annotation with table and row, we have the same annotation three times. In fact, a similar query on a table with n columns, would produce $n + 1$ annotations. Looking at the trace expression below, we can see the problem.

```
for (x <- table "xs" ⟨a: Int, b: Bool⟩)
  [For ⟨in=⟨a=Cell ⟨tbl="xs", col="a", row=x.oid, val=x.a⟩,
            b=Cell ⟨tbl="xs", col="b", row=x.oid, val=x.b⟩⟩,
       out=Cell ⟨tbl="xs", col="a", row=x.oid, val=x.a⟩⟩]
```

The record case combines the annotations from all of the fields, which interacts badly with the tracing of tables, which puts annotations on all of the fields. There are at least two solutions to this problem that preserve tracing at the level of cells. The ad-hoc solution is to introduce a set union operator $M \cup N$ with a special normalization rule that reduces to just M if M and N are known to be equal statically. The proper solution would be to support set and multiset semantics for different portions of the same query and generate SQL queries that eliminate duplicates where necessary.

4.4 Normalization and query generation

To compute the where-provenance of the earlier boat tour agencies query (let's call it Q), we can specialize the `wherep` trace analysis function to the traced type of Q and apply it to the traced query itself as follows:

```
wherep (TRACE [⟨name:String,phone:String⟩]) (trace Q)
```

We have seen that traces can get quite big and trace analysis functions contain features with no obvious counterpart in SQL. The rest of this paper shows how exactly tracing works, describes the language in detail, and discusses normalization to nested relational calculus, which we can further translate to SQL. In the end, all of the trace construction and trace analysis code will be eliminated and the above code will result in a simple query like the following.

```
SELECT e.name AS name_val, 'externalTours' AS name_tbl,
       'name' AS name_col, e.oid AS name_row,
       a.phone AS phone_val, 'agencies' AS phone_tbl,
       'phone' AS phone_col, a.oid AS phone_row
  FROM agencies AS a, externaltours AS e
 WHERE a.name = e.name AND e.type = 'boat'
```

Note that Links flattens nested records into top-level columns and only reassembles records when fetching the results [9].

5 Links^T syntax & static semantics

The syntax of Links^T is summarized in Figure 5. Links^T is a simplification of the core language for Links queries introduced by Lindley and Cheney [18]. Links employs row typing to typecheck record expressions; *row variables* can be used to quantify over parts of record types. The core Links calculus of [18] also covers ordinary Links code and the type-and-effect system used to ensure query expressions only perform operations that are possible on the database. We omit these aspects as well as more recent extensions such as algebraic effects and handlers [17] and session types [19].

In addition to the core query language constructs, Links^T draws heavily on the λ_i^{ML} calculus [16], which supports *intensional polymorphism*, that is, the capability to analyze types at run time (**typecase**) and define types by recursion on the structure of other types (**Typerec**). Analogous capabilities are also provided for rows, similar to the *type-level record computation* used in Ur/Web [10].

We use a single context Γ for both type variables α and term variables x. In addition to the usual kinds *Type* and $\to$, we have *Row*, the kind of rows. We distinguish type and row constructors from types and rows (again following λ_i^{ML}). The difference is that constructors can be subject to type analysis (e.g. **typecase**), and can contain type-level computation (e.g. **Typerec**), but unlike types, cannot employ polymorphism. Constructors include base type constructors, type variables (we write ρ for type variables with kind *Row*), type-level functions and application, list, record, and trace type constructors, as well as **Typerec** to analyze type constructors. Types do not include any computation, but constructors can be embedded into types using $\mathsf{T}(C)$. More often than not, types and constructors are either equivalent or it is obvious from the context which we are talking about, so we will write, e.g., `Bool` to mean either the type, or the constructor `Bool`*. We write $[A]$ and $[C]$ for list types and constructors and $\langle R \rangle$ and $\langle S \rangle$ for record types and constructors.

Because type constructors can contain nontrivial computation due to **Typerec**, **Rmap** and type-level lambda-abstraction, Links^T employs equivalence judgments for types, rows, and their constructors. The more interesting of the type-level computation rules are shown in Figure 6. The full set of equivalence rules and type-level computation rules are relegated to in the appendix due to space limitations. We conjecture that type equivalence and typechecking are decidable for Links^T (they are for λ_i^{ML}) but this remains to be fully investigated.

Most of the typing rules are standard. The more interesting rules can be found in Figure 7. We require that all tables have an `oid` column and otherwise only contain fields of base types. We can map a sufficiently polymorphic function over a record using **rmap**. This is reflected on the type level with the row type constructor **Rmap**. We can fold a homogeneous record into a single value using **rfold**. Note that we do not specify the order of folding, so it is best to use a commutative combining function. The rule for **typecase** is standard, but the improved rule by Crary et al. [13] would work as well.

The most representative introduction and elimination rules for the `Trace` type can be found in Figure 8. The constructors for comprehensions and polymorphic operators carry type information. This type information is brought back in scope when analyzing traces using **typecase**: the respective branches bind both a type and a term variable.

6 The self-tracing transformation

The self-tracing transformation turns a *normalized* query expression into an expression that produces a trace of its own execution. As seen in Figure 9, most cases are straightforward. Variables inside a self-tracing query refer to their subtrace directly. Tables are the only source of `Cell` trace constructors. Comprehensions and conditionals need to distribute a trace constructor over a subtrace of any shape including lists and record types. We accomplish this with the meta-level helper function *dist*. It takes a type, an expression with a hole $\mathbb{H}$ in it, and a value of the given type and traverses lists and records until it reaches the leaves and wraps the expression with the hole around them. Alternatively, we could have written *dist* as a Links function with the type

```
dist: ∀a. (∀b. Trace b -> Trace b) -> TRACE a -> TRACE a
```

but using it requires a lot of boilerplate code for handling impossible cases, so we prefer the definition in Figure 9.

With these definitions in hand, we check that the self-tracing transformation preserves well-formedness. Note that the type-level function `TRACE` is needed to state these properties. Proof details are in the appendix.

Lemma 10. *For all types C that can appear in query types (base types, list types, closed record types), all expressions k with a hole $\mathbb{H}$ that have type* `Trace` *D assuming the hole $\mathbb{H}$ has type* `Trace` *D, and all expressions M of type* `TRACE` *C, dist*$(\mathsf{TRACE}\ C, k, M)$ *has type* `TRACE` *C.*

Theorem 11. *If $\Gamma \vdash M : A$ then for all C, if $\Gamma \vdash A = \mathsf{T}(C)$ then $[\![\Gamma]\!] \vdash [\![M]\!] : \mathsf{T}(\mathsf{TRACE}\ C)$, where Γ is a context that maps all term variables to closed records with fields of base type and M is a plain Links query term in normal form.*

7 Normalization

Our ultimate goal is to translate Links^T queries — including provenance extraction by trace analysis — to Sql. We know from previous work [9, 12, 18, 29] that Nrc expressions, extended with sum types and higher-order functions, can be translated to Sql as long as their return type is nested relational. In this section, we extend query normalization to deal with the new features for tracing and trace analysis.

Contexts	Γ	::=	$\cdot \mid \Gamma, \alpha : K \mid \Gamma, x : A$
Kinds	K	::=	$\mathit{Type} \mid \mathit{Row} \mid K_1 \to K_2$
Constructors	C, D	::=	$\texttt{Bool}^* \mid \texttt{Int}^* \mid \texttt{String}^* \mid \alpha \mid \lambda\alpha : K.C \mid C\ D \mid \texttt{List}^*\ C \mid \texttt{Record}^*\ S \mid \texttt{Trace}^*\ C$
		$\mid$	$\textbf{Typerec}\ C\ (C_B, C_I, C_S, C_L, C_R, C_T)$
Row Constructors	S	::=	$\cdot \mid l : C; S \mid \rho \mid \textbf{Rmap}\ C\ S$
Types	A, B	::=	$\texttt{T}(C) \mid \texttt{Bool} \mid \texttt{Int} \mid \texttt{String} \mid A \to B \mid \texttt{List}\ A \mid \texttt{Record}\ R \mid \texttt{Trace}\ A \mid \forall\alpha : K.A$
Rows	R	::=	$\cdot \mid l : A; R$
Expressions	L, M, N	::=	$c \mid x \mid \lambda x : A.M \mid M\ N \mid \Lambda\alpha : K.M \mid M\ C \mid \textbf{fix}\ f : A.M$
		$\mid$	$\textbf{if}\ L\ \textbf{then}\ M\ \textbf{else}\ N \mid M + N \mid M == N \mid \langle\rangle \mid \langle l = M; N\rangle \mid M.l$
(Collections)		$\mid$	$[] \mid [M] \mid M +\!\!+ N \mid \textbf{for}\ (x \leftarrow M)\ N \mid \textbf{table}\ n\ \langle R\rangle$
(Traces)		$\mid$	$\texttt{Lit}\ M \mid \texttt{If}\ M \mid \texttt{For}\ C\ M \mid \texttt{Cell}\ M \mid \texttt{OpEq}\ C\ M \mid \texttt{OpPlus}\ M$
(Trace Analysis)		$\mid$	$\textbf{tracecase}\ M\ \textbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P)$
(Type Analysis)		$\mid$	$\textbf{typecase}\ C\ \textbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \beta.M_T) \mid \textbf{rmap}^S\ L\ M \mid \textbf{rfold}^S\ L\ M\ N$

Figure 5. The syntax of Links$^{\mathrm{T}}$.

$$
\begin{aligned}
(\lambda\alpha : K.C)\ D &\rightsquigarrow C[\alpha := D] \\
\textbf{Rmap}\ C\ \cdot &\rightsquigarrow \cdot \\
\textbf{Rmap}\ C\ (l : D; S) &\rightsquigarrow (l : C\ D; \textbf{Rmap}\ C\ S) \\
\textbf{Typerec}\ \texttt{Bool}\ (C_B, \ldots) &\rightsquigarrow C_B \\
\textbf{Typerec}\ [D]\ (\ldots, C_L, \ldots) &\rightsquigarrow C_L\ D\ (\textbf{Typerec}\ D\ (\ldots, C_L, \ldots)) \\
\textbf{Typerec}\ \langle S\rangle\ (\ldots, C_R, \ldots) &\rightsquigarrow \\
&\quad C_R\ S\ (\textbf{Rmap}\ (\lambda\alpha.\textbf{Typerec}\ \alpha\ (\ldots, C_R, \ldots))\ S)
\end{aligned}
$$

Figure 6. Constructor and row constructor computation.

We show progress and preservation which imply the existence of a partial normalization function. Unlike standard progress and preservation, we do not normalize to values, but to a normal form that includes table references and residual query code which is ultimately translated to Sql queries.

We cannot show strong normalization, since we require recursive functions to be able to analyze arbitrary queries.

7.1 Reduction rules

Links$^{\mathrm{T}}$ uses the same general approach to normalization as plain Links [9]. We define a relation $\rightsquigarrow$ between terms. Most rules are standard. Figure 12 shows the β-rules for the new Links$^{\mathrm{T}}$ features. Since constructors can appear in terms, e.g., typecase, we also need to normalize constructors. We use the same rules as for type-level computation (Figure 6). We also need to add commuting conversions to, e.g., lift if-then-else out of tracecase, to expose additional β reductions. The full rules can be found in the appendix in Figures 26, 31, 32, 33.

Unlike plain Links, we allow recursion in queries and unroll fixpoints as necessary. It is up to the programmer to ensure that their functions terminate. Record map and record fold inspect their row constructor argument only. Record map evaluates to a new record where we apply the given function to each field's type and value. Record fold applies the given function to the accumulator and every record field's value successively. We evaluate tracecase and typecase by reducing to the appropriate branch and substituting terms and constructors for term and type variables.

7.2 Preservation

To prove preservation we will need several substitution lemmas. Substitution of variables in terms, type variables in types, and type variables in terms are standard for λ_i^{ML} [13, 21]. We additionally need variants for row constructors: substitution of row variables in types and substitution of row variables in terms. We also need standard context manipulation lemmas for weakening and swapping the order of unrelated variables. For details, see Appendix C.1.

Now we can prove that the reduction relation $\rightsquigarrow$ preserves the kinds of constructors and the types of terms.

Lemma 13. *For all type constructors C and row constructors S, contexts Γ, and kinds K, if $\Gamma \vdash C : K$ and $C \rightsquigarrow C'$, then $\Gamma \vdash C' : K$ and if $\Gamma \vdash S : K$ and $S \rightsquigarrow S'$, then $\Gamma \vdash S' : K$.*

The proof is straightforward by induction on the kinding derivation. For details, see Section C.6.

Lemma 14 (Preservation)**.** *For all terms M and M', contexts Γ, and types A, if $\Gamma \vdash M : A$ and $M \rightsquigarrow M'$, then $\Gamma \vdash M' : A$.*

The proof is by induction on the typing derivation $\Gamma \vdash M : A$. The cases for record map and record fold require type equivalence under type-level computation. The cases for typecase require the more exotic substitution lemmas from before. See Section C.7 for the proof.

7.3 Normal form

The goal of normalization is to perform partial evaluation of those parts of the program that are independent of database

$$\frac{\Gamma \vdash M : B \qquad \Gamma \vdash A = B}{\Gamma \vdash M : A} \qquad \frac{\cdot \vdash R : \mathit{Row}}{\Gamma \vdash \mathbf{table}\ n\ \langle \mathsf{oid} : \mathtt{Int}; R\rangle : [\langle \mathsf{oid} : \mathtt{Int}; R\rangle]} \qquad \frac{\Gamma \vdash M : \forall \alpha : \mathit{Type}.\mathtt{T}(\alpha) \to \mathtt{T}(C\ \alpha) \qquad \Gamma \vdash N : \mathtt{T}(\mathtt{Record}^*\ S)}{\Gamma \vdash \mathbf{rmap}^S\ M\ N : \mathtt{T}(\mathtt{Record}^*\ (\mathbf{Rmap}\ C\ S))}$$

$$\frac{\Gamma \vdash L : \mathtt{T}(C) \to \mathtt{T}(C) \to \mathtt{T}(C) \qquad \Gamma \vdash M : \mathtt{T}(C) \qquad \Gamma \vdash N : \mathtt{T}(\mathtt{Record}^*\ (\mathbf{Rmap}\ (\lambda\alpha.\alpha \to C)\ S))}{\Gamma \vdash \mathbf{rfold}^S\ L\ M\ N : \mathtt{T}(C)}$$

$$\frac{\begin{array}{c}\Gamma \vdash C : \mathit{Type} \\ \Gamma, \alpha : \mathit{Type} \vdash B : \mathit{Type} \qquad \beta, \rho, \gamma \notin \mathit{Dom}(\Gamma) \qquad \Gamma \vdash M_B : B[\alpha := \mathtt{Bool}^*] \qquad \Gamma \vdash M_I : B[\alpha := \mathtt{Int}^*] \qquad \Gamma \vdash M_S : B[\alpha := \mathtt{String}^*] \\ \Gamma, \beta : \mathit{Type} \vdash M_L : B[\alpha := \mathtt{List}^*\ \beta] \qquad \Gamma, \rho : \mathit{Row} \vdash M_R : B[\alpha := \mathtt{Record}^*\ \rho] \qquad \Gamma, \gamma : \mathit{Type} \vdash M_T : B[\alpha := \mathtt{Trace}^*\ \gamma]\end{array}}{\Gamma \vdash \mathbf{typecase}\ C\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) : B[\alpha := C]}$$

Figure 7. Term formation $\Gamma \vdash M : A$.

$$\frac{\Gamma \vdash c : \mathtt{Int}}{\Gamma \vdash \mathtt{Lit}\ c : \mathtt{Trace\ Int}} \qquad \frac{\Gamma \vdash M : \langle \mathsf{cond} : \mathtt{Trace\ Bool}, \mathsf{out} : \mathtt{Trace}\ A\rangle}{\Gamma \vdash \mathtt{If}\ M : \mathtt{Trace}\ A} \qquad \frac{\Gamma \vdash C : \mathit{Type} \qquad \Gamma \vdash M : \langle \mathsf{in} : \mathtt{T(TRACE}\ C), \mathsf{out} : \mathtt{Trace}\ A\rangle}{\Gamma \vdash \mathtt{For}\ C\ M : \mathtt{Trace}\ A}$$

$$\frac{\Gamma \vdash M : \langle \mathsf{tbl} : \mathtt{String}, \mathsf{col} : \mathtt{String}, \mathsf{row} : \mathtt{Int}, \mathsf{val} : A\rangle}{\Gamma \vdash \mathtt{Cell}\ M : \mathtt{Trace}\ A} \qquad \frac{\Gamma \vdash C : \mathit{Type} \qquad \Gamma \vdash M : \langle \mathsf{l} : \mathtt{T(TRACE}\ C), \mathsf{r} : \mathtt{T(TRACE}\ C)\rangle}{\Gamma \vdash \mathtt{OpEq}\ C\ M : \mathtt{Trace\ Bool}}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \mathtt{Trace}\ A \qquad \Gamma, x_L : A \vdash M_L : B \qquad \Gamma, x_I : \langle \mathsf{cond} : \mathtt{Trace\ Bool}, \mathsf{then} : \mathtt{Trace}\ A\rangle \vdash M_I : B \\ \Gamma, \alpha_F : \mathit{Type}, x_F : \langle \mathsf{in} : \mathtt{T(TRACE}\ \alpha_F), \mathsf{out} : \mathtt{Trace}\ A\rangle \vdash M_F : B \qquad \Gamma, x_C : \langle \mathsf{tbl} : \mathtt{String}, \mathsf{col} : \mathtt{String}, \mathsf{row} : \mathtt{Int}, \mathsf{val} : A\rangle \vdash M_C : B \\ \Gamma, \alpha_E : \mathit{Type}, x_E : \langle \mathsf{l} : \mathtt{T(TRACE}\ \alpha_E), \mathsf{r} : \mathtt{T(TRACE}\ \alpha_E)\rangle \vdash M_E : B \qquad \Gamma, x_P : \langle \mathsf{l} : \mathtt{Trace\ Int}, \mathsf{r} : \mathtt{Trace\ Int}\rangle \vdash M_P : B\end{array}}{\Gamma \vdash \mathbf{tracecase}\ M\ \mathbf{of}\ (x_L.M_L, x_I.M_I, \alpha_F.x_F.M_F, x_C.M_C, \alpha_E.x_E.M_E, x_P.M_P) : B}$$

Figure 8. `Trace` introduction and elimination rules (some `Lit` cases and `OpPlus` omitted).

values. In particular, we look to eliminate all language constructs which we cannot translate to Sql. The Links$^{\mathrm{T}}$ normal form (Figure 15) describes what terms look like after exhaustive application of the rewriting rules. It appears we were not successful, seeing that record map and fold, tracecase, and typecase are all still present. However, the normal form grammar splits constructors into normal constructors C and neutral constructors E, and row constructors into normal row constructors S and neutral row constructors U.

Remark 16. *Neutral constructors E and neutral row constructors U always contain at least one free type variable α or ρ and those are the only base cases for their respective sort.*

We will later use the above to show that some term forms are impossible within queries. Queries do not contain free type variables, so E and U collapse into nothing, and terms built from E and U (like **rmap**) cannot appear.

Similarly, terms are split into normal terms M and neutral terms F. The latter are stuck on a free variable x, a stuck constructor E, or a stuck row constructor U. We will later argue that inside a query all variables are references to tables and therefore restricted to be base types or records with fields of base types. This means they cannot be functions or trace constructors and therefore record map, record fold, tracecase, and typecase do not actually appear in normal form queries.

7.4 Progress

Progress states that well typed terms either already are in the normal form described in the previous section or that there is a further reduction step possible. Reduction preserves typing, so we can keep reducing until we reach normal form and thus obtain a partial normalization function.

Like preservation, progress is split into two lemmas: one for constructors and row constructors and one for terms.

Lemma 17. *All well-kinded type constructors C and row constructors S, are either in normal form, or there is a type constructor C' with $C \rightsquigarrow C'$, or row constructor S' with $S \rightsquigarrow S'$.*

The proof is straightforward by induction on the kinding derivations of C and S (see Section C.8).

Lemma 18 (Progress)**.** *For all well-typed terms M, either M is in normal form, or there is a term M' with $M \rightsquigarrow M'$.*

The proof (see Section C.9) is by induction on the typing derivation of M. Most nontrivial cases have three parts: reduce in subterms via congruence rules; a β-rule applies; or a commuting conversion applies.

7.5 Normal terms with query types are NRC

Links$^{\mathrm{T}}$ normal form still includes language constructs such as typecase, which do not have an obvious Sql counterpart. In this section, we will argue that these cannot actually occur

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![c]\!] &= \mathtt{Lit}\ c \\
[\![M + N]\!] &= \mathtt{OpPlus}\ \langle \mathtt{l} = [\![M]\!], \mathtt{r} = [\![N]\!] \rangle \\
[\![M == (N : \mathtt{T}(C))]\!] &= \mathtt{OpEq}\ C\ \langle \mathtt{l} = [\![M]\!], \mathtt{r} = [\![N]\!] \rangle \\
[\![\langle \overline{l = M} \rangle]\!] &= \langle \overline{l = [\![M]\!]} \rangle \\
[\![M.l]\!] &= [\![M]\!].l \\
[\![[]]\!] &= [] \\
[\![[M]]\!] &= [[\![M]\!]] \\
[\![M +\!\!+ N]\!] &= [\![M]\!] +\!\!+ [\![N]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{table}\ n\ \langle \overline{l : C} \rangle]\!] = \mathbf{for}\ (y \leftarrow \mathbf{table}\ n\ \langle \overline{l : C} \rangle) \\
[\langle \overline{l = \mathtt{Cell}\langle \mathtt{tbl} = n, \mathtt{col} = l, \mathtt{row} = y.\mathtt{oid}, \mathtt{val} = y.l \rangle} \rangle]
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{for}\ (x \leftarrow M : D)\ N : \mathtt{T}(C)]\!] = \mathbf{for}\ (x \leftarrow [\![M]\!]) \\
\mathit{dist}(\mathtt{TRACE}\ C, \mathtt{For}\ D\ \langle \mathtt{in} = x, \mathtt{out} = \mathbb{H} \rangle, [\![N]\!])
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N : \mathtt{T}(C)]\!] = \mathbf{if}\ \mathtt{value}\ (\mathtt{Trace}\ \mathtt{Bool})\ [\![L]\!] \\
\mathbf{then}\ \mathit{dist}(\mathtt{TRACE}\ C, \mathtt{If}\langle \mathtt{cond} = [\![L]\!], \mathtt{out} = \mathbb{H} \rangle, [\![M]\!]) \\
\mathbf{else}\ \mathit{dist}(\mathtt{TRACE}\ C, \mathtt{If}\langle \mathtt{cond} = [\![L]\!], \mathtt{out} = \mathbb{H} \rangle, [\![N]\!])
\end{aligned}
$$

$$
\begin{aligned}
\mathit{dist}(\langle \overline{l : C} \rangle, k, r) &= \langle \overline{l = \mathit{dist}(C, k, r.l)} \rangle \\
\mathit{dist}([C], k, l) &= \mathbf{for}\ (x \leftarrow l)\ [\mathit{dist}(C, k, x)] \\
\mathit{dist}(\mathtt{Trace}\ C, k, t) &= k[\mathbb{H} := t]
\end{aligned}
$$

Figure 9. The self-tracing transformation.

$$
\begin{aligned}
\mathbf{fix}\ f.M &\rightsquigarrow M[f := \mathbf{fix}\ f.M] \\
(\Lambda\alpha.M)\ C &\rightsquigarrow M[\alpha := C] \\
\mathbf{rmap}^{(\overline{l_i : C_i})}\ M\ N &\rightsquigarrow \langle \overline{l_i = (M\ C_i)\ N.l_i} \rangle \\
\mathbf{rfold}^{(\overline{l_i : C_i})}\ L\ M\ N &\rightsquigarrow L\ N.l_1\ (L\ N.l_2 \ldots (L\ N.l_n\ M) \ldots) \\
\mathbf{tracecase}\ \mathtt{Lit}\ M\ \mathbf{of}\ (x.M_L, \ldots) &\rightsquigarrow M_L[x := M] \\
\mathbf{tracecase}\ \mathtt{For}\ C\ M\ \mathbf{of}\ (\ldots, \alpha x.M_F, \ldots) &\rightsquigarrow M_F[\alpha := C, x := M] \\
\mathbf{typecase}\ \mathtt{Bool}\ \mathbf{of}\ (M_B, \ldots) &\rightsquigarrow M_B \\
\mathbf{typecase}\ [C]\ \mathbf{of}\ (\ldots, \beta.M_L, \ldots) &\rightsquigarrow M_L[\beta := C] \\
\mathbf{typecase}\ \langle S \rangle\ \mathbf{of}\ (\ldots, \rho.M_R, \ldots) &\rightsquigarrow M_R[\rho := S]
\end{aligned}
$$

Figure 12. Normalization β-rules.

in a query. Queries are closed expressions with nested relational type. Inside a query, all variables refer to tables. This is captured in the following definition of query contexts.

Definition 19 (Query context)**.**

- The empty context $\cdot$ is a query context.
- The context $\Gamma, x : \langle \overline{l_i : A_i} \rangle$ is a query context, if Γ is a query context, x is not bound in Γ already, and each type A_i is a base type.

The Links$^{\mathrm{T}}$ normal form includes neutral terms F, which include record map and fold, tracecase, and typecase. With the following Lemma, we will further restrict which terms F can appear in queries to just variables x and projections $x.l$.

Lemma 20. *A term in neutral form F that is well-typed in a query context Γ, is of the form x or $x.l$.*

Proof. By induction on the typing derivation. The term cannot be a record fold or typecase, because those necessarily contain a (row) type variable (Remark 16), which is unbound in the query context Γ (Definition 19). It cannot be a term application, type application, or tracecase, because the term in function position or the scrutinee, by IH, is of the form x or $x.l$, both of which are ill-typed given that the query context Γ does not contain function types, polymorphic types, or trace types. Projections $P.l$ are of the form $F.l$ or $(\mathbf{rmap}^U\ M\ N).l$. The former case reduces by IH to $x.l$ or $x.l'.l$, the first of which is okay, and the second is ill-typed. The latter case is impossible, because U necessarily contains a row variable and would therefore be ill-typed. This leaves variables x and projections of variables $x.l$. □

Finally, we can use this to show that query terms in Links$^{\mathrm{T}}$ normal form are actually in nested relational calculus already.

Theorem 22. *If M is a term in normal form with a nested relational type in a query context Γ, then M is in the nested relational calculus (Figure 21).*

The proof (Section C.11) is by induction on the typing derivation, making use of query contexts (Definition 19), Remark 16, and Lemma 20.

From here, we can use previous work such as query shredding [9] or flattening [28] to produce Sql.

8 Related work

Extracting provenance from traces is not a new idea [2, 7, 22]. What makes our work different is that traces and trace analysis are defined in the language itself. In combination with query normalization, this makes Links$^{\mathrm{T}}$ the first, to our knowledge, system that can execute user-defined query trace analysis on the database.

The traces in Links$^{\mathrm{T}}$ take inspiration from work on *slicing* of database queries and programs [7, 23, 24]. Compared to theirs, our traces contain less information. Some information would be easy to add, like concatenation operations or projections. Other information requires changing the structure of traces in a more invasive way. In particular, our traces are cell-level only and do not include information about the binding structure of queries. We also trace only after a first normalization phase, so traces do not include information about, e.g., functions in the original query code. Expression-shaped traces with explicit representation of variables like those proposed by Cheney et al. [7], seem to make writing well-typed analysis functions more difficult.

Normal constructors	C	::=	$E \mid \texttt{Bool}^* \mid \texttt{Int}^* \mid \texttt{String}^* \mid \lambda\alpha : K.C \mid \texttt{List}^*\ C \mid \texttt{Record}^*\ S \mid \texttt{Trace}^*\ C$
Neutral constructors	E	::=	$\alpha \mid E\ C \mid \textbf{Typerec}\ E\ (C_B, C_I, C_S, C_L, C_R, C_T)$
Normal row constructors	S	::=	$U \mid \cdot \mid l : C; S$
Neutral row constructors	U	::=	$\rho \mid l : C; U \mid \textbf{Rmap}\ C\ U$
Normal terms	M, N	::=	$F \mid c \mid \lambda x : A.M \mid \Lambda\alpha : K.M \mid \textbf{if}\ H\ \textbf{then}\ M\ \textbf{else}\ N \mid M + N \mid \langle\rangle \mid \langle l = M; N\rangle$
		|	$[] \mid [M] \mid M +\!\!+ N \mid \textbf{for}\ (x \leftarrow T)\ N \mid \textbf{table}\ n\ \langle R\rangle$
		|	$\texttt{Lit}\ M \mid \texttt{If}\ M \mid \texttt{For}\ C\ M \mid \texttt{Cell}\ M \mid \texttt{OpEq}\ C\ M \mid \texttt{OpPlus}\ M$
Neutral terms	F	::=	$x \mid P.l \mid F\ M \mid F\ C \mid \textbf{rfold}^U\ L\ M\ N \mid \textbf{rmap}^U\ M\ N$
		|	$\textbf{tracecase}\ F\ \textbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P)$
		|	$\textbf{typecase}\ E\ \textbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \beta.M_T)$
Neutral conditional	H	::=	$F \mid M == N$
Neutral projection	P	::=	$F \mid \textbf{rmap}^U\ M\ N$
Neutral table	T	::=	$F \mid \textbf{table}\ n\ \langle R\rangle$

Figure 15. Links$^{\mathrm{T}}$ normal form.

Types	A	::=	$\texttt{Bool} \mid \texttt{Int} \mid \texttt{String} \mid [A] \mid \langle\overline{l : A}\rangle$
Terms	M, N, L	::=	$c \mid x \mid \langle\overline{l = M}\rangle \mid M.l \mid M + N \mid M == N$
		|	$\textbf{if}\ L\ \textbf{then}\ M\ \textbf{else}\ N \mid \textbf{table}\ n\ \langle\overline{l : A}\rangle$
		|	$[] \mid [M] \mid M +\!\!+ N \mid \textbf{for}\ (x \leftarrow N)\ M$

Figure 21. Target normal form for queries: Nrc.

Müller et al. [22] trace query execution and show how non-standard interpretations of the Sql semantics produce where-provenance and lineage instead of query results. They decompose traces into a static part that resembles the shape of the query, and a dynamic part which records control-flow decisions made by the database during query execution. Their work extends to Sql features like grouping and aggregation that are not implemented in Links, let alone traced in Links$^{\mathrm{T}}$. Unlike in Links$^{\mathrm{T}}$, alternative interpretation of queries happens after a trace has been recorded. Thus it is not possible for the database to optimize, for example, filters based on provenance information.

Links$^{\mathrm{T}}$ builds on λ_i^{ML} [21]. The λ_r calculus of Crary et al. [13] improves on λ_i^{ML} in making runtime type information explicit, avoiding passing types where unnecessary, and improving the ergonomics of the typecase typing rule by refining types in context. An actual implementation would benefit from these improvements.

Links$^{\mathrm{T}}$ features generic record programming in the form of record mapping and folding. Ur/Web [11] features "first class, type-level names and records" [10]. Its generic and metaprogramming features seem suitable for our needs, but Ur/Web currently lacks the advanced query normalization features we require. Type inference for Links$^{\mathrm{T}}$ is an open problem. Type inference for Ur/Web is undecidable. However, Chlipala [10] claims that heuristics work well-enough in practice to mostly avoid proof terms and complex type annotations. Maybe this could be a model for Links$^{\mathrm{T}}$, too.

While we present this work as an extension of Links and its query normalization rules, it is conceivable that one could similarly extend other systems such as the flattening transformation implemented in Dsh [28], or the tagless final implementation of query shredding by Suzuki et al. [27].

9 Conclusions

Language-integrated support for queries and their provenance seems promising, but currently requires nontrivial interventions in the language implementation or sophisticated metaprogramming capabilities. In this paper, we take a step towards making language-integrated provenance easily customizable by factoring provenance translations into a self-tracing transformation (that can be implemented once and for all) and generic programming and trace analysis capabilities (that can be used to implement different provenance transformations). Nevertheless, our work so far is a foundational language design and more remains to be done to make it practical. We have not said anything about typechecking or inference or, more generally, how Links$^{\mathrm{T}}$ interfaces with the rest of Links. The expressiveness and generality of our approach to traces needs to be tested further, by using it to implement other forms of provenance. Conversely, the features of Links$^{\mathrm{T}}$ may have further applications beyond provenance, like the row-generic programming techniques employed by Ur/Web. In particular, even without traces and trace analysis, our results extend the theory of conservativity for Nrc queries to normalization of typecase and typerec constructs (albeit in the presence of nonterminating fixedpoint computations). Sharpening these results to ensure termination of trace analysis functions would also be an interesting challenge.

Acknowledgments This work was supported by a Google Faculty Research Award and ERC Consolidator Grant Skye (grant number 682315).

References

[1] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2012. A Core Calculus for Provenance. In *Principles of Security and Trust*. Springer, 410–429.

[2] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2013. A core calculus for provenance. *Journal of Computer Security* 21 (2013), 919–969. https://doi.org/10.3233/JCS-130487 Full version of a POST 2012 paper.

[3] Bahareh Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. 2014. A Generic Provenance Middleware for Queries, Updates, and Transactions. In *6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014)*. USENIX Association. https://www.usenix.org/conference/tapp2014/agenda/presentation/arab

[4] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT 2001 (LNCS)*. Springer, 316–330. https://doi.org/10.1007/3-540-44503-X_20

[5] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of Programming with Complex Objects and Collection Types. *Theor. Comp. Sci.* 149, 1 (1995), 3–48.

[6] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. In *ACM SIGPLAN International Conference on Functional Programming*. https://doi.org/10.1145/1086365.1086397

[7] James Cheney, Amal Ahmed, and Umut A. Acar. 2014. Database Queries that Explain their Work. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014)*. ACM, 271–282. https://doi.org/10.1145/2643135.2643143

[8] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (April 2009), 379–474. https://doi.org/10.1561/1900000006

[9] James Cheney, Sam Lindley, and Philip Wadler. 2014. Query Shredding: Efficient Relational Evaluation of Queries over Nested Multisets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD 2014)*. ACM, 1027–1038. https://doi.org/10.1145/2588555.2612186

[10] Adam Chlipala. 2010. Ur: Statically-typed Metaprogramming with Type-level Record Computation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010)*. ACM, 122–133. https://doi.org/10.1145/1806596.1806612

[11] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*. ACM, 153–165. https://doi.org/10.1145/2676726.2677004

[12] Ezra Cooper. 2009. The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. In *DBPL 2009*. LNCS, Vol. 5708. Springer, 36–51. https://doi.org/10.1007/978-3-642-03793-1_3

[13] Karl Crary, Stephanie Weirich, and Greg Morrisett. 2002. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming* 12, 6 (2002), 567–600. https://doi.org/10.1017/S0956796801004282

[14] Stefan Fehrenbach and James Cheney. 2018. Language-integrated provenance. *Science of Computer Programming* 155 (2018), 103 – 145. https://doi.org/10.1016/j.scico.2017.08.009 Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2016.

[15] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 31–40. https://doi.org/10.1145/1265530.1265535

[16] Robert Harper and Greg Morrisett. 1995. Compiling Polymorphism Using Intensional Type Analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995)*. ACM, 130–141. https://doi.org/10.1145/199448.199475

[17] Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016)*. ACM, New York, NY, USA, 15–27. https://doi.org/10.1145/2976022.2976033

[18] Sam Lindley and James Cheney. 2012. Row-based Effect Types for Database Integration. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2012)*. ACM, 91–102. https://doi.org/10.1145/2103786.2103798

[19] Sam Lindley and J. Garrett Morris. 2017. Lightweight functional session types. In *Behavioural Types: from Theory to Tools*. River Publishers. https://doi.org/10.13052/rp-9788793519817

[20] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD 2006)*. ACM, 706–706. https://doi.org/10.1145/1142473.1142552

[21] Greg Morrisett. 1995. *Compiling with types*. Ph.D. Dissertation. Carnegie Mellon University. https://www.cs.cmu.edu/~rwh/theses/morrisett.pdf

[22] Tobias Müller, Benjamin Dietrich, and Torsten Grust. 2018. You Say 'What', I Hear 'Where' and 'Why': (Mis-)Interpreting SQL to Derive Fine-grained Provenance. *Proceedings of the VLDB Endowment* 11, 11 (July 2018), 1536–1549. https://doi.org/10.14778/3236187.3236204

[23] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional Programs that Explain their Work. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*. ACM, 365–376. https://doi.org/10.1145/2364527.2364579

[24] Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. 2017. Imperative Functional Programs That Explain Their Work. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 14 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110258

[25] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: Provenance and Probability Management in PostgreSQL. *Proceedings of the VLDB Endowment* 11, 12 (Aug. 2018), 2034–2037. https://doi.org/10.14778/3229863.3236253

[26] Jan Stolarek and James Cheney. 2018. Language-integrated provenance in Haskell. *The Art, Science, and Engineering of Programming* 2, 3 (4 2018). https://doi.org/10.22152/programming-journal.org/2018/2/11

[27] Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Finally, Safely-extensible and Efficient Language-integrated Query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2016)*. ACM, 37–48. https://doi.org/10.1145/2847538.2847542

[28] Alexander Ulrich and Torsten Grust. 2015. The Flatter, the Better: Query Compilation Based on the Flattening Transformation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 2015)*. ACM, 1421–1426. https://doi.org/10.1145/2723372.2735359

[29] Limsoon Wong. 1996. Normal Forms and Conservative Extension Properties for Query Languages over Collection Types. *J. Comput. System Sci.* 52, 3 (1996), 495 – 505. https://doi.org/10.1006/jcss.1996.0037

$$\frac{}{\cdot \text{ is well-formed}} \qquad \frac{\Gamma \vdash A : \mathit{Type} \qquad x \notin \mathit{Dom}(\Gamma)}{\Gamma, x : A \text{ is well-formed}} \qquad \frac{\Gamma \text{ is well-formed} \qquad \alpha \notin \mathit{Dom}(\Gamma)}{\Gamma, \alpha : K \text{ is well-formed}}$$

Figure 23. Well-formed contexts Γ.

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{Bool}^* : \mathit{Type}} \quad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{Int}^* : \mathit{Type}} \quad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{String}^* : \mathit{Type}} \quad \frac{\Gamma(\alpha) = K}{\Gamma \vdash \alpha : K} \quad \frac{\Gamma, \alpha : K_1 \vdash C : K_2}{\Gamma \vdash \lambda \alpha : K_1 . C : K_1 \to K_2} \quad \frac{\Gamma \vdash C : K_1 \to K_2 \qquad \Gamma \vdash D : K_1}{\Gamma \vdash C\ D : K_2}$$

$$\frac{\Gamma \vdash C : \mathit{Type}}{\Gamma \vdash \mathtt{List}^*\ C : \mathit{Type}} \qquad \frac{\Gamma \vdash S : \mathit{Row}}{\Gamma \vdash \mathtt{Record}^*\ S : \mathit{Type}} \qquad \frac{\Gamma \vdash C : \mathit{Type}}{\Gamma \vdash \mathtt{Trace}^*\ C : \mathit{Type}}$$

$$\frac{\Gamma \vdash C : \mathit{Type} \quad \Gamma \vdash C_B : K \quad \Gamma \vdash C_I : K \quad \Gamma \vdash C_S : K \quad \Gamma \vdash C_L : \mathit{Type} \to K \to K \quad \Gamma \vdash C_R : \mathit{Row} \to \mathit{Row} \to K \quad \Gamma \vdash C_T : \mathit{Type} \to K \to K}{\Gamma \vdash \mathbf{Typerec}\ C\ (C_B, C_I, C_S, C_L, C_R, C_T) : K}$$

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash \cdot : \mathit{Row}} \qquad \frac{\Gamma \vdash C : \mathit{Type} \qquad \Gamma \vdash S : \mathit{Row}}{\Gamma \vdash l : C; S : \mathit{Row}} \qquad \frac{\Gamma \vdash C : \mathit{Type} \to \mathit{Type} \qquad \Gamma \vdash S : \mathit{Row}}{\Gamma \vdash \mathbf{Rmap}\ C\ S : \mathit{Row}}$$

Figure 24. Constructor and row constructor kinding.

$$\frac{\Gamma \vdash C : \mathit{Type}}{\Gamma \vdash \mathtt{T}(C) : \mathit{Type}} \quad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{Bool} : \mathit{Type}} \quad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{Int} : \mathit{Type}} \quad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{String} : \mathit{Type}} \quad \frac{\Gamma, \alpha : K \vdash A : \mathit{Type} \qquad \alpha \notin \mathit{Dom}(\Gamma)}{\Gamma \vdash \forall \alpha : K . A : \mathit{Type}}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash B : \mathit{Type}}{\Gamma \vdash A \to B : \mathit{Type}} \quad \frac{\Gamma \vdash A : \mathit{Type}}{\Gamma \vdash \mathtt{List}\ A : \mathit{Type}} \quad \frac{\Gamma \vdash R : \mathit{Row}}{\Gamma \vdash \mathtt{Record}\ R : \mathit{Type}} \quad \frac{\Gamma \vdash A : \mathit{Type}}{\Gamma \vdash \mathtt{Trace}\ A : \mathit{Type}} \quad \frac{\Gamma \vdash S : \mathit{Row}}{\Gamma \vdash \mathtt{T}(S) : \mathit{Row}} \quad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \cdot : \mathit{Row}}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash R : \mathit{Row}}{\Gamma \vdash l : A; R : \mathit{Row}}$$

Figure 25. Type and row type kinding.

A The value trace analysis function

```
VALUE = λa:Type.Typerec a (Bool, Int, String, λ_ b.List b, λ_ r.Record r, λc _.c)

value : ∀a.T(a) -> T(VALUE a)
value = fix (value: ∀a.T(a) -> T(VALUE a)).Λa:Type.
  typecase a of
    Bool     => λx:Bool.x
    Int      => λx:Int.x
    String   => λx:String.x
    List b   => λx:List b.for (y <- x) [value b y]
    Record r => λx:Record r.rmap^r value x
    Trace b  => λx:Trace b.tracecase x of
      Lit y    => y
      If y     => value (Trace b) y.out
      For c y  => value (Trace b) y.out
      Cell y   => y.data
      OpPlus y => value (Trace Int) y.left + value (Trace Int) y.right
      OpEq c y => value (TRACE c) y.left == value (TRACE c) y.right
```

B Full formalization of Links^{T}

B.1 Kinding judgments

- Figure 23 gives the rules for well-typed contexts ($\Gamma, \alpha : K$ is well-formed)
- Figure 24 defines the well-formedness judgment for type constructors ($\Gamma \vdash C : K$)
- Figure 25 defines the well-formedness judgment for types ($\Gamma \vdash A : K$)

$$S \rightsquigarrow S' \Rightarrow l : C; S \rightsquigarrow l : C; S'$$
$$C \rightsquigarrow C' \Rightarrow l : C; S \rightsquigarrow l : C'; S$$

$$C \rightsquigarrow C' \Rightarrow C\ D \rightsquigarrow C'\ D$$
$$D \rightsquigarrow D' \Rightarrow C\ D \rightsquigarrow C\ D'$$
$$(\lambda\alpha : K.C)\ D \rightsquigarrow C[\alpha := D]$$

$$C \rightsquigarrow C' \Rightarrow \lambda\alpha : K.C \rightsquigarrow \lambda\alpha : K.C'$$
$$C \rightsquigarrow C' \Rightarrow \mathtt{List}^*\ C \rightsquigarrow \mathtt{List}^*\ C'$$
$$C \rightsquigarrow C' \Rightarrow \mathtt{Trace}^*\ C \rightsquigarrow \mathtt{Trace}^*\ C'$$
$$S \rightsquigarrow S' \Rightarrow \mathtt{Record}^*\ S \rightsquigarrow \mathtt{Record}^*\ S'$$

$$\mathbf{Rmap}\ C\ \cdot \rightsquigarrow \cdot$$
$$\mathbf{Rmap}\ C\ (l : D; S) \rightsquigarrow (l : C\ D; \mathbf{Rmap}\ C\ S)$$
$$S \rightsquigarrow S' \Rightarrow \mathbf{Rmap}\ C\ S \rightsquigarrow \mathbf{Rmap}\ C\ S'$$
$$C \rightsquigarrow C' \Rightarrow \mathbf{Rmap}\ C\ S \rightsquigarrow \mathbf{Rmap}\ C'\ S$$

$$C \rightsquigarrow C' \Rightarrow \mathbf{Typerec}\ C\ (C_B, C_I, C_S, C_L, C_R, C_T) \rightsquigarrow \mathbf{Typerec}\ C'\ (C_B, C_I, C_S, C_L, C_R, C_T)$$
$$C_B \rightsquigarrow C'_B \Rightarrow \mathbf{Typerec}\ C\ (C_B, C_I, C_S, C_L, C_R, C_T) \rightsquigarrow \mathbf{Typerec}\ C\ (C'_B, C_I, C_S, C_L, C_R, C_T)$$
$$\vdots$$
$$\mathbf{Typerec}\ \mathtt{Bool}^*\ (C_B, C_I, C_S, C_L, C_R, C_T) \rightsquigarrow C_B$$
$$\mathbf{Typerec}\ \mathtt{Int}^*\ (C_B, C_I, C_S, C_L, C_R, C_T) \rightsquigarrow C_I$$
$$\mathbf{Typerec}\ \mathtt{String}^*\ (C_B, C_I, C_S, C_L, C_R, C_T) \rightsquigarrow C_S$$
$$\mathbf{Typerec}\ \mathtt{List}^*\ D\ (C_B, C_I, C_S, C_L, C_R, C_T) \rightsquigarrow C_L\ D\ (\mathbf{Typerec}\ D\ (C_B, C_I, C_S, C_L, C_R, C_T))$$
$$\mathbf{Typerec}\ \mathtt{Record}^*\ S\ (C_B, C_I, C_S, C_L, C_R, C_T) \rightsquigarrow C_R\ S\ (\mathbf{Rmap}\ (\lambda\alpha.\mathbf{Typerec}\ \alpha\ (C_B, C_I, C_S, C_L, C_R, C_T))\ S)$$
$$\mathbf{Typerec}\ \mathtt{Trace}^*\ D\ (C_B, C_I, C_S, C_L, C_R, C_T) \rightsquigarrow C_T\ D\ (\mathbf{Typerec}\ D\ (C_B, C_I, C_S, C_L, C_R, C_T))$$

Figure 26. Constructor and row constructor computation.

B.2 Type-level computation and equivalence

- Figure 26 defines the reduction relation for type and row constructors ($C \rightsquigarrow C'$, $S \rightsquigarrow S'$)
- Figure 27 defines equivalence for type and row constructors ($\Gamma \vdash C = C' : K$, $\Gamma \vdash S = S' : K$)
- Figure 28 defines type and row equivalence ($\Gamma \vdash A = B : K$, $\Gamma \vdash S = S' : \mathit{Type}$)

B.3 Type judgments

- Figure 29 defines the typing judgment for most of the Links$^{\mathsf{T}}$ constructs ($\Gamma \vdash M : A$)
- Figure 30 defines the typing rules introducing and eliminating traces.

B.4 Normalization

- Figure 31 defines the main computational rules (β-rules) for normalization ($M \rightsquigarrow M'$)
- Figure 32 defines commuting conversion rules for normalization ($M \rightsquigarrow M'$)
- Figure 33 defines congruence rules for normalization ($M \rightsquigarrow M'$)

C Proofs

C.1 Additional properties

Besides the properties stated in the main body of the paper, the following additional properties are needed:

Lemma 34 (Substitution lemmas)**.**

1. *If* $\Gamma, x : A \vdash M : B$ *and* $\Gamma \vdash N : A$ *then* $\Gamma \vdash M[x := N] : B$.

$$\frac{\Gamma \vdash C : K}{\Gamma \vdash C = C : K} \qquad \frac{\Gamma \vdash D = C : K}{\Gamma \vdash C = D : K} \qquad \frac{\Gamma \vdash C = C' : K \qquad \Gamma \vdash C' = C'' : K}{\Gamma \vdash C = C'' : K} \qquad \frac{\Gamma \vdash C : K \to K'}{\Gamma \vdash \lambda\alpha : K.C\ \alpha = C : K \to K'}$$

$$\frac{\Gamma, \alpha : K \vdash C = D : K' \qquad \alpha \notin Dom(\Gamma)}{\Gamma \vdash \lambda\alpha : K.C = \lambda\alpha : K.D : K \to K'} \qquad \frac{\Gamma \vdash C = C' : K' \to K \qquad \Gamma \vdash D = D' : K'}{\Gamma \vdash C\ D = C'\ D' : K} \qquad \frac{\Gamma \vdash C = D : K}{\Gamma \vdash \mathtt{List}^*\ C = \mathtt{List}^*\ D : K}$$

$$\frac{\Gamma \vdash S = S' : K}{\Gamma \vdash \mathtt{Record}^*\ S = \mathtt{Record}^*\ S' : K} \qquad \frac{\Gamma \vdash C = D : K}{\Gamma \vdash \mathtt{Trace}^*\ C = \mathtt{Trace}^*\ D : K} \qquad \frac{\Gamma \vdash C : K \qquad \Gamma \vdash D : K \qquad C \rightsquigarrow D}{\Gamma \vdash C = D : K} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \cdot = \cdot : Row}$$

$$\frac{\Gamma \vdash C = D : Type \qquad \Gamma \vdash S = S' : Row}{\Gamma \vdash (l : C; S) = (l : D; S') : Row} \qquad \frac{\Gamma \vdash C = D : Type \to Type \qquad \Gamma \vdash S = S' : Row}{\Gamma \vdash \mathbf{Rmap}\ C\ S = \mathbf{Rmap}\ D\ S' : Row}$$

$$\frac{\begin{array}{c}\Gamma \vdash C = C' : K \qquad \Gamma \vdash C_B = C'_B : K \\ \Gamma \vdash C_I = C'_I : K \qquad \Gamma \vdash C_S = C'_S : K \qquad \Gamma \vdash C_L = C'_L : Type \to K \to K \qquad \Gamma \vdash C_R = C'_R : Row \to Row \to K \qquad \Gamma \vdash C_T = C'_T : Type \to K \to K\end{array}}{\Gamma \vdash \mathbf{Typerec}\ C\ (C_B, C_I, C_S, C_L, C_R, C_T) = \mathbf{Typerec}\ C'\ (C'_B, C'_I, C'_S, C'_L, C'_R, C'_T) : K}$$

Figure 27. Constructor and row constructor equivalence.

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{T}(\mathtt{Bool}^*) = \mathtt{Bool} : Type} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{T}(\mathtt{Int}^*) = \mathtt{Int} : Type} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{T}(\mathtt{String}^*) = \mathtt{String} : Type} \qquad \frac{\Gamma \vdash C : Type}{\Gamma \vdash \mathtt{T}(\mathtt{List}^*\ C) = \mathtt{List}\ \mathtt{T}(C) : Type}$$

$$\frac{\Gamma \vdash S : Row}{\Gamma \vdash \mathtt{T}(\mathtt{Record}^*\ S) = \mathtt{Record}\ \mathtt{T}(S) : Type} \qquad \frac{\Gamma \vdash C : Type}{\Gamma \vdash \mathtt{T}(\mathtt{Trace}^*\ C) = \mathtt{Trace}\ \mathtt{T}(C) : Type} \qquad \frac{}{\Gamma \vdash T(\cdot) = \cdot : Row} \qquad \frac{\Gamma \vdash C : Type \qquad \Gamma \vdash S : Row}{\Gamma \vdash T(l : C; S) = (l : T(C); T(S)) : Row}$$

$$\frac{\Gamma \vdash C = D : Type}{\Gamma \vdash \mathtt{T}(C) = \mathtt{T}(D) : Type} \qquad \frac{\Gamma \vdash S = S' : Row}{\Gamma \vdash \mathtt{T}(S) = \mathtt{T}(S') : Row} \qquad \frac{\Gamma \vdash A = B : Type}{\Gamma \vdash \mathtt{List}\ A = \mathtt{List}\ B : Type} \qquad \frac{\Gamma \vdash R = R' : Row}{\Gamma \vdash \mathtt{Record}\ R = \mathtt{Record}\ R' : Type}$$

$$\frac{\Gamma \vdash A = B : Type}{\Gamma \vdash \mathtt{Trace}\ A = \mathtt{Trace}\ B : Type} \qquad \frac{\Gamma \vdash A = A' : Type \qquad \Gamma \vdash B = B' : Type}{\Gamma \vdash A \to B = A' \to B' : Type} \qquad \frac{\Gamma \vdash A = B : Type}{\Gamma \vdash \forall\alpha.A = \forall\alpha.B : Type} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \cdot = \cdot : Row}$$

$$\frac{\Gamma \vdash A = B : Type \qquad \Gamma \vdash R = R' : Row}{\Gamma \vdash (l : A; R) = (l : B; R') : Row}$$

Figure 28. Type and row type equivalence.

2. *If* $\Gamma, \alpha : K \vdash A : K'$ *and* $\Gamma \vdash C : K$ *then* $\Gamma[\alpha := C] \vdash A[\alpha := C] : K'[\alpha := C]$.
3. *If* $\Gamma, \rho : K \vdash A : K'$ *and* $\Gamma \vdash S : K$ *then* $\Gamma[\rho := S] \vdash A[\rho := S] : K'[\rho := S]$.
4. *If* $\Gamma, \alpha : K \vdash M : A$ *and* $\Gamma \vdash C : K$ *then* $\Gamma[\alpha := C] \vdash M[\alpha := C] : A[\alpha := C]$.
5. *If* $\Gamma, \rho : K \vdash M : A$ *and* $\Gamma \vdash S : K$ *then* $\Gamma[\rho := S] \vdash M[\rho := S] : A[\rho := S]$.

Lemma 35 (Weakening). *If* $\Gamma \vdash M : A$, $\Gamma \vdash B : K$, *and* x *does not appear free in* Γ, M, A, *then* $\Gamma, x : B \vdash M : A$.

Lemma 36 (Context swap)**.**

1. *If* $\Gamma, x : A_x, y : A_y \vdash M : B$ *then* $\Gamma, y : A_y, x : A_x \vdash M : B$.
2. *If* $\Gamma, x : A_x, y : A_y \vdash B : K_B$ *then* $\Gamma, y : A_y, x : A_x \vdash B : K_B$.
3. *If* $\Gamma, \alpha : K_\alpha, y : A_y \vdash M : B$ *and* α *does not appear free in* A_y *then* $\Gamma, y : A_y, \alpha : K_\alpha \vdash M : B$.
4. *If* $\Gamma, \alpha : K_\alpha, y : A_y \vdash B : K_B$ *and* α *does not appear free in* A_y *then* $\Gamma, y : A_y, \alpha : K_\alpha \vdash B : K_B$.
5. *If* $\Gamma, x : A_x, \beta : K_\beta \vdash M : B$ *then* $\Gamma, \beta : K_\beta, x : A_x, \vdash M : B$.
6. *If* $\Gamma, x : A_x, \beta : K_\beta \vdash B : K_B$ *then* $\Gamma, \beta : K_\beta, x : A_x, \vdash B : K_B$.
7. *If* $\Gamma, \alpha : K_\alpha \beta : K_\beta \vdash M : B$ *and* α *does not appear free in* K_β *then* $\Gamma, \beta : K_\beta, \alpha : K_\alpha \vdash M : B$.
8. *If* $\Gamma, \alpha : K_\alpha \beta : K_\beta \vdash B : K_B$ *and* α *does not appear free in* K_β *then* $\Gamma, \beta : K_\beta, \alpha : K_\alpha \vdash B : K_B$.

Lemma 37. *For all query type constructors* C *and row constructors* S *and well-formed contexts* Γ*:*

$$\Gamma \vdash \mathsf{VALUE}(\mathsf{TRACE}\ C) = C$$

and

$$\Gamma \vdash \mathbf{Rmap}\ \mathsf{VALUE}\ (\mathbf{Rmap}\ \mathsf{TRACE}\ S) = S$$

$$\frac{\Sigma(c) = A}{\Gamma \vdash c : A} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash M : B \qquad x \notin \mathit{Dom}(\Gamma)}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\frac{\Gamma, \alpha : K \vdash M : A \qquad \alpha \notin \mathit{Dom}(\Gamma)}{\Gamma \vdash \Lambda\alpha : K.M : \forall \alpha : K.A} \qquad \frac{\Gamma \vdash M : \forall \alpha : K.A \qquad \Gamma \vdash C : K}{\Gamma \vdash M\ C : A[\alpha := C]} \qquad \frac{\Gamma \vdash A \qquad \Gamma, f : A \vdash M : A}{\Gamma \vdash \mathbf{fix}\ f : A.M : A} \qquad \frac{\Gamma \vdash L : \mathsf{Bool} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash \mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N : A}$$

$$\frac{\Gamma \vdash M : \mathsf{Int} \qquad \Gamma \vdash N : \mathsf{Int}}{\Gamma \vdash M + N : \mathsf{Int}} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A \qquad \Gamma \vdash A : \mathit{Type}}{\Gamma \vdash M == N : \mathsf{Bool}} \qquad \frac{\cdot \vdash R : \mathit{Row}}{\Gamma \vdash \mathbf{table}\ n\ \langle \mathsf{oid} : \mathsf{Int}, R\rangle : \mathsf{List}\langle \mathsf{oid} : \mathsf{Int}; R\rangle} \qquad \frac{\Gamma \vdash A : \mathit{Type}}{\Gamma \vdash [] : \mathsf{List}\ A}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : \mathsf{List}\ A} \qquad \frac{\Gamma \vdash M : \mathsf{List}\ A \qquad \Gamma \vdash N : \mathsf{List}\ A}{\Gamma \vdash M +\!\!+ N : \mathsf{List}\ A} \qquad \frac{\Gamma \vdash M : \mathsf{List}\ A \qquad \Gamma, x : A \vdash N : \mathsf{List}\ B}{\Gamma \vdash \mathbf{for}\ (x \leftarrow M)\ N : \mathsf{List}\ B} \qquad \frac{\Gamma\ \text{well-formed}}{\Gamma \vdash \langle\rangle : \mathsf{Record}\ ()}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : \mathsf{Record}\ R}{\Gamma \vdash \langle l = M; N\rangle : \mathsf{Record}\ (l : A; R)} \qquad \frac{\Gamma \vdash M : \mathsf{Record}\ (l : A; R)}{\Gamma \vdash M.l : A} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A = B}{\Gamma \vdash M : A}$$

$$\frac{\Gamma \vdash M : \forall \alpha : \mathit{Type}.\mathsf{T}(\alpha) \to \mathsf{T}(C\ \alpha) \qquad \Gamma \vdash N : \mathsf{T}(\mathsf{Record}^*\ S)}{\Gamma \vdash \mathbf{rmap}^S\ M\ N : \mathsf{T}(\mathsf{Record}^*\ (\mathbf{Rmap}\ C\ S))}$$

$$\frac{\Gamma \vdash L : \mathsf{T}(C) \to \mathsf{T}(C) \to \mathsf{T}(C) \qquad \Gamma \vdash M : \mathsf{T}(C) \qquad \Gamma \vdash N : \mathsf{T}(\mathsf{Record}^*\ (\mathbf{Rmap}\ (\lambda\alpha.\alpha \to C)\ S))}{\Gamma \vdash \mathbf{rfold}^S\ L\ M\ N : \mathsf{T}(C)}$$

$$\frac{\begin{array}{c}\Gamma \vdash C : \mathit{Type} \qquad \Gamma, \alpha : \mathit{Type} \vdash B : \mathit{Type} \qquad \beta, \rho, \gamma \notin \mathit{Dom}(\Gamma) \qquad \Gamma \vdash M_B : B[\alpha := \mathsf{Bool}^*] \qquad \Gamma \vdash M_I : B[\alpha := \mathsf{Int}^*] \\ \Gamma \vdash M_S : B[\alpha := \mathsf{String}^*] \qquad \Gamma, \beta : \mathit{Type} \vdash M_L : B[\alpha := \mathsf{List}^*\ \beta] \qquad \Gamma, \rho : \mathit{Row} \vdash M_R : B[\alpha := \mathsf{Record}^*\ \rho] \qquad \Gamma, \gamma : \mathit{Type} \vdash M_T : B[\alpha := \mathsf{Trace}^*\ \gamma]\end{array}}{\Gamma \vdash \mathbf{typecase}\ C\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) : B[\alpha := C]}$$

Figure 29. Term formation $\Gamma \vdash M : A$.

$$\frac{\Gamma \vdash c : \mathsf{Bool}}{\Gamma \vdash \mathsf{Lit}\ c : \mathsf{Trace\ Bool}} \qquad \frac{\Gamma \vdash c : \mathsf{Int}}{\Gamma \vdash \mathsf{Lit}\ c : \mathsf{Trace\ Int}} \qquad \frac{\Gamma \vdash c : \mathsf{String}}{\Gamma \vdash \mathsf{Lit}\ c : \mathsf{Trace\ String}} \qquad \frac{\Gamma \vdash M : \langle \mathsf{cond} : \mathsf{Trace\ Bool}, \mathsf{out} : \mathsf{Trace}\ A\rangle}{\Gamma \vdash \mathsf{If}\ M : \mathsf{Trace}\ A}$$

$$\frac{\Gamma \vdash C : \mathit{Type} \qquad \Gamma \vdash M : \langle \mathsf{in} : \mathsf{T}(\mathsf{TRACE}\ C), \mathsf{out} : \mathsf{Trace}\ A\rangle}{\Gamma \vdash \mathsf{For}\ C\ M : \mathsf{Trace}\ A} \qquad \frac{\Gamma \vdash M : \langle \mathsf{table} : \mathsf{String}, \mathsf{column} : \mathsf{String}, \mathsf{row} : \mathsf{Int}, \mathsf{data} : A\rangle}{\Gamma \vdash \mathsf{Cell}\ M : \mathsf{Trace}\ A}$$

$$\frac{\Gamma \vdash C : \mathit{Type} \qquad \Gamma \vdash M : \langle \mathsf{left} : \mathsf{T}(\mathsf{TRACE}\ C), \mathsf{right} : \mathsf{T}(\mathsf{TRACE}\ C)\rangle}{\Gamma \vdash \mathsf{OpEq}\ C\ M : \mathsf{Trace\ Bool}} \qquad \frac{\Gamma \vdash M : \langle \mathsf{left} : \mathsf{Trace\ \ Int}, \mathsf{right} : \mathsf{Trace\ \ Int}\rangle}{\Gamma \vdash \mathsf{OpPlus}\ M : \mathsf{Trace\ Int}}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \mathsf{Trace}\ A \qquad \Gamma, x_L : A \vdash M_L : B \qquad \Gamma, x_I : \langle \mathsf{cond} : \mathsf{Trace\ Bool}, \mathsf{then} : \mathsf{Trace}\ A\rangle \vdash M_I : B \\ \Gamma, \alpha_F : \mathit{Type}, x_F : \langle \mathsf{in} : \mathsf{T}(\mathsf{TRACE}\ \alpha_F), \mathsf{out} : \mathsf{Trace}\ A\rangle \vdash M_F : B \qquad \Gamma, x_C : \langle \mathsf{table} : \mathsf{String}, \mathsf{column} : \mathsf{String}, \mathsf{row} : \mathsf{Int}, \mathsf{data} : A\rangle \vdash M_C : B \\ \Gamma, \alpha_E : \mathit{Type}, x_E : \langle \mathsf{left} : \mathsf{T}(\mathsf{TRACE}\ \alpha_E), \mathsf{right} : \mathsf{T}(\mathsf{TRACE}\ \alpha_E)\rangle \vdash M_E : B \qquad \Gamma, x_P : \langle \mathsf{left} : \mathsf{Trace\ Int}, \mathsf{right} : \mathsf{Trace\ Int}\rangle \vdash M_P : B\end{array}}{\Gamma \vdash \mathbf{tracecase}\ M\ \mathbf{of}\ (x_L.M_L, x_I.M_I, \alpha_F.x_F.M_F, x_C.M_C, \alpha_E.x_E.M_E, x_P.M_P) : B}$$

Figure 30. Trace introduction and elimination rules.

Lemma 38. *For all query types C,* TRACE C *is not a base type.*

Definition 39 (Trace context)**.** $[\![\Gamma]\!]$ maps term variable x to $\mathsf{T}(\mathsf{TRACE}\ C)$ if and only if Γ maps x to A, where C is the obvious constructor with $\cdot \vdash A = \mathsf{T}(C)$.

Lemma 40. *For every query type A made of base types, list constructors, and closed records, there exists C such that $\Gamma \vdash A = \mathsf{T}(C)$ in a well-formed context Γ.*

C.2 Proof of Lemma 37

Proof. By induction on query types C and closed rows of query types S.

- Base types Bool^*, Int^*, String^*:

$$\mathsf{VALUE}(\mathsf{TRACE\ Bool}^*) = \mathsf{VALUE}(\mathsf{Trace\ Bool}^*) = \mathsf{Bool}^*$$

$$
\begin{aligned}
(\lambda x.M)\ N &\rightsquigarrow M[x \coloneqq N] \\
\mathbf{fix}\ f.M &\rightsquigarrow M[f \coloneqq \mathbf{fix}\ f.M] \\
(\Lambda \alpha.M)\ C &\rightsquigarrow M[\alpha \coloneqq C] \\
\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ M\ \mathbf{else}\ N &\rightsquigarrow M \\
\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ M\ \mathbf{else}\ N &\rightsquigarrow N \\
\langle \overline{l_i = M_i} \rangle.l_i &\rightsquigarrow M_i \\
\mathbf{rmap}^{(\overline{l_i : C_i})}\ M\ N &\rightsquigarrow \langle \overline{l_i = (M\ C_i)\ N.l_i} \rangle \\
\mathbf{rfold}^{(\overline{l_i : C_i})}\ L\ M\ N &\rightsquigarrow L\ N.l_1\ (L\ N.l_2 \ldots (L\ N.l_n\ M) \ldots) \\
\mathbf{for}\ (x \leftarrow [])\ N &\rightsquigarrow [] \\
\mathbf{for}\ (x \leftarrow [M])\ N &\rightsquigarrow N[x \coloneqq M] \\
\mathbf{tracecase}\ \mathtt{Lit}\ M\ \mathbf{of}\ (x.M_L, M_I, M_F, M_C, M_E, M_P) &\rightsquigarrow M_L[x \coloneqq M] \\
\mathbf{tracecase}\ \mathtt{If}\ M\ \mathbf{of}\ (M_L, x.M_I, M_F, M_C, M_E, M_P) &\rightsquigarrow M_I[x \coloneqq M] \\
\mathbf{tracecase}\ \mathtt{For}\ C\ M\ \mathbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P) &\rightsquigarrow M_F[\alpha \coloneqq C, x \coloneqq M] \\
\mathbf{tracecase}\ \mathtt{Cell}\ M\ \mathbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P) &\rightsquigarrow M_C[x \coloneqq M] \\
\mathbf{tracecase}\ \mathtt{OpEq}\ C\ M\ \mathbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P) &\rightsquigarrow M_E[\alpha \coloneqq C, x \coloneqq M] \\
\mathbf{tracecase}\ \mathtt{OpPlus}\ M\ \mathbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P) &\rightsquigarrow M_P[x \coloneqq M] \\
\mathbf{typecase}\ \mathtt{Bool}\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) &\rightsquigarrow M_B \\
\mathbf{typecase}\ \mathtt{Int}\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) &\rightsquigarrow M_I \\
\mathbf{typecase}\ \mathtt{String}\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) &\rightsquigarrow M_S \\
\mathbf{typecase}\ \mathtt{List}\ C\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) &\rightsquigarrow M_L[\beta \coloneqq C] \\
\mathbf{typecase}\ \mathtt{Record}\ S\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) &\rightsquigarrow M_R[\rho \coloneqq S] \\
\mathbf{typecase}\ \mathtt{Trace}\ C\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) &\rightsquigarrow M_T[\gamma \coloneqq C]
\end{aligned}
$$

Figure 31. Normalization β-rules. See also commuting conversions in Figure 32, congruence rules in Figure 33, and constructor computation rules in Figure 26

$$
\begin{aligned}
(\mathbf{if}\ L\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2)\ N &\rightsquigarrow \mathbf{if}\ L\ \mathbf{then}\ M_1\ N\ \mathbf{else}\ M_2\ N \\
(\mathbf{if}\ L\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2)\ C &\rightsquigarrow \mathbf{if}\ L\ \mathbf{then}\ M_1\ C\ \mathbf{else}\ M_2\ C \\
(\mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N).l &\rightsquigarrow \mathbf{if}\ L\ \mathbf{then}\ M.l\ \mathbf{else}\ N.l \\
\mathbf{for}\ (x \leftarrow M_1 +\!\!+ M_2)\ N &\rightsquigarrow (\mathbf{for}\ (x \leftarrow M_1)\ N) +\!\!+ (\mathbf{for}\ (x \leftarrow M_2)\ N) \\
\mathbf{for}\ (x \leftarrow \mathbf{for}\ (y \leftarrow L)\ M)\ N &\rightsquigarrow \mathbf{for}\ (y \leftarrow L)\ \mathbf{for}\ (x \leftarrow M)\ N \\
\mathbf{if}\ (\mathbf{if}\ L\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2)\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2 &\rightsquigarrow \mathbf{if}\ L\ \mathbf{then}\ (\mathbf{if}\ M_1\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2)\ \mathbf{else}\ (\mathbf{if}\ M_2\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2) \\
\mathbf{for}\ (x \leftarrow \mathbf{if}\ L\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2)\ N &\rightsquigarrow \mathbf{if}\ L\ \mathbf{then}\ \mathbf{for}\ (x \leftarrow M_1)\ N\ \mathbf{else}\ \mathbf{for}\ (x \leftarrow M_2)\ N \\
\mathbf{tracecase}\ \mathbf{if}\ L\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2\ \mathbf{of}\ (M_L, M_I, M_F, M_C, M_E, M_P) &\rightsquigarrow \mathbf{if}\ L\ \mathbf{then}\ \mathbf{tracecase}\ M_1\ \mathbf{of}\ (M_L, M_I, M_F, M_C, M_E, M_P) \\
&\qquad \mathbf{else}\ \mathbf{tracecase}\ M_2\ \mathbf{of}\ (M_L, M_I, M_F, M_C, M_E, M_P)
\end{aligned}
$$

Figure 32. Commuting conversions reorder expressions to expose more β-reductions.

- List types $\mathtt{List}^*\ D$:

$$
\begin{aligned}
\mathsf{VALUE}(\mathsf{TRACE}\ (\mathtt{List}^*\ D)) &= \mathsf{VALUE}(\mathtt{List}^*\ (\mathsf{TRACE}\ D)) \\
&= \mathtt{List}^*\ (\mathsf{VALUE}(\mathsf{TRACE}\ D)) \\
&= \mathtt{List}^*\ D
\end{aligned}
$$

$$\frac{M \rightsquigarrow M'}{I[M] \rightsquigarrow I[M']} \qquad \frac{C \rightsquigarrow C'}{J[C] \rightsquigarrow J[C']}$$

Term frames $I[]$ ::=
$\lambda x.[] \mid [] \; N \mid M \; [] \mid \Lambda\alpha.[] \mid [] \; C \mid \texttt{if} \; [] \; \texttt{then} \; M \; \texttt{else} \; N \mid \texttt{if} \; L \; \texttt{then} \; [] \; \texttt{else} \; N \mid \texttt{if} \; L \; \texttt{then} \; M \; \texttt{else} \; []$
$\mid \langle l = []; N\rangle \mid \langle l = M; []\rangle \mid [].l \mid \texttt{rmap}^S \; [] \; N \mid \texttt{rmap}^S \; M \; [] \mid \texttt{rfold}^S \; [] \; M \; N \mid \texttt{rfold}^S \; L \; [] \; N \mid \texttt{rfold}^S \; L \; M \; []$
$\mid \texttt{[[]]} \mid [] +\!\!+ N \mid M +\!\!+ N \mid [] == N \mid M == N \mid [] + N \mid M + N \mid \texttt{for} \; (x \leftarrow []) \; N \mid \texttt{for} \; (x \leftarrow M) \; []$
$\mid \texttt{Lit} \; [] \mid \texttt{If} \; [] \mid \texttt{For} \; C \; [] \mid \texttt{Cell} \; [] \mid \texttt{OpEq} \; C \; [] \mid \texttt{OpPlus} \; []$
$\mid \texttt{tracecase} \; [] \; \texttt{of} \; (M_L, M_I, M_F, M_C, M_E, M_P) \mid \texttt{tracecase} \; M \; \texttt{of} \; ([], M_I, M_F, M_C, M_E, M_P)$
$\mid \texttt{tracecase} \; M \; \texttt{of} \; (M_L, [], M_F, M_C, M_E, M_P) \mid \texttt{tracecase} \; M \; \texttt{of} \; (M_L, M_I, [], M_C, M_E, M_P)$
$\mid \texttt{tracecase} \; M \; \texttt{of} \; (M_L, M_I, M_F, [], M_E, M_P) \mid \texttt{tracecase} \; M \; \texttt{of} \; (M_L, M_I, M_F, M_C, [], M_P)$
$\mid \texttt{tracecase} \; M \; \texttt{of} \; (M_L, M_I, M_F, M_C, M_E, [])$
$\mid \texttt{typecase} \; C \; \texttt{of} \; ([], M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T)$
$\mid \texttt{typecase} \; C \; \texttt{of} \; (M_B, [], M_S, \beta.M_L, \rho.M_R, \gamma.M_T) \mid \texttt{typecase} \; C \; \texttt{of} \; (M_B, M_I, [], \beta.M_L, \rho.M_R, \gamma.M_T)$
$\mid \texttt{typecase} \; C \; \texttt{of} \; (M_B, M_I, M_S, \beta.[], \rho.M_R, \gamma.M_T) \mid \texttt{typecase} \; C \; \texttt{of} \; (M_B, M_I, M_S, \beta.M_L, \rho.[], \gamma.M_T)$
$\mid \texttt{typecase} \; C \; \texttt{of} \; (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.[])$

Constructor frames $J[]$ ::=
$M \; [] \mid \texttt{rmap}^{[]} \; M \; N \mid \texttt{rfold}^{[]} \; L \; M \; N \mid \texttt{For} \; [] \; M \mid \texttt{OpEq} \; [] \; M$
$\mid \texttt{typecase} \; [] \; \texttt{of} \; (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T)$

Figure 33. Congruence rules allow subterms to reduce independently.

- Record types $\texttt{Record}^* \; S$:

$$\begin{aligned}\texttt{VALUE}(\texttt{TRACE} \; (\texttt{Record}^* \; S)) &= \texttt{VALUE}(\texttt{Record}^*(\textbf{Rmap} \; \texttt{TRACE} \; S)) \\ &= \texttt{Record}^* \; (\textbf{Rmap} \; \texttt{VALUE} \; (\textbf{Rmap} \; \texttt{TRACE} \; S)) \\ &= \texttt{Record}^* \; S\end{aligned}$$

- Empty row $\cdot$: $\textbf{Rmap} \; \texttt{VALUE} \; (\textbf{Rmap} \; \texttt{TRACE} \; \cdot) = \cdot$
- Row cons $(l : A, S)$:

$$\begin{aligned}&\textbf{Rmap} \; \texttt{VALUE} \; (\textbf{Rmap} \; \texttt{TRACE} \; (l : A, S)) \\ =&\textbf{Rmap} \; \texttt{VALUE} \; (l : \texttt{TRACE} \; A, \textbf{Rmap} \; \texttt{TRACE} \; S) \\ =&(l : \texttt{VALUE} \; (\texttt{TRACE} \; A), \textbf{Rmap} \; \texttt{VALUE} \; (\textbf{Rmap} \; \texttt{TRACE} \; S)) \\ =&(l : A, \textbf{Rmap} \; \texttt{VALUE} \; (\textbf{Rmap} \; \texttt{TRACE} \; S)) \\ =&(l : A, S)\end{aligned}$$

□

C.3 Proof of Lemma 38

Proof. By induction on query types C made up from base types, lists, and closed records. Applying TRACE to base types `Bool`, `Int`, and `String` results in traced base types `Trace Bool`, `Trace Int`, and `Trace String`, respectively. List types are guarded by the `List` type constructor, and similarly for records. Traces are not query types, but if they were, the induction hypothesis would apply. □

C.4 Proof of Lemma 10

Proof. By induction on the query type C.

- The base cases are `Bool`, `Int`, and `String`. For any base type O out of these, we have $\texttt{TRACE} \; O = \texttt{Trace} \; O$. We have $dist(\texttt{Trace} \; O, k, t) = k[\mathbb{H} := t]$ and need to show that it has type `Trace` O. Both t and $\mathbb{H}$ have type `Trace` O, so substituting one for the other in k does not change the type (Lemma 34).
- Case $C = \texttt{List} \; (\texttt{TRACE} \; C')$: We need the right-hand side $\textbf{for} \; (x \leftarrow l) \; [dist(\texttt{TRACE} \; C', k, x)]$ to have type $\texttt{TRACE} \; (\texttt{List} \; C')$. We use the rules for comprehension and singleton list. We now need to show that $dist(\texttt{TRACE} \; C', k, x)$ has type $\texttt{TRACE} \; C'$ which is true by induction hypothesis with the same k.
- Case $C = \langle \overline{l : \texttt{TRACE} \; C'} \rangle$: The right-hand side $\langle \overline{l = dist(\texttt{TRACE} \; C', k, r.l)} \rangle$ needs to have type $\langle \overline{l : \texttt{TRACE} \; C'} \rangle$. Thus, by record construction and record projection, we need each of the expressions $dist(\texttt{TRACE} \; C', k, r.l)$ to have type $\texttt{TRACE} \; C'$ which they do by induction hypothesis. □

C.5 Proof of Theorem 11

Proof. By induction on the typing derivation for $M : \mathsf{T}(C)$. Almost all cases require that some subterms have a type $\mathsf{T}(C')$ that is equal to some query type A. We can obtain this constructor C' by Lemma 40.

- Case $\dfrac{\Gamma(x) = A}{\Gamma \vdash x : A}$: $\dfrac{[\![\Gamma]\!](x) = \mathsf{T}(\mathsf{TRACE}\ C) \quad \text{(Definition 39)}}{[\![\Gamma]\!] \vdash x : \mathsf{T}(\mathsf{TRACE}\ C)}$
- Literals c have base types `Bool`, `Int`, or `String`. Their traces `Lit` c have types `Trace Bool`, `Trace Int`, or `Trace String`, respectively.
- Case $\dfrac{\Gamma \vdash L : \mathsf{Bool} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash \mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N : A}$:
  The right hand side of the self-tracing transform is another **`if-then-else`** with condition `value (Trace Bool)` $[\![L]\!]$ and then-branch
  $$\mathit{dist}(\mathsf{TRACE}\ C, \mathtt{If}\ \langle \mathtt{cond} = [\![L]\!], \mathtt{out} = \mathbb{H}\rangle, [\![M]\!])$$
  and similar else-branch.
  In the condition, we apply $\mathtt{value} : \forall\alpha.\mathsf{T}(\alpha) \to \mathsf{T}(\mathsf{VALUE}\ \alpha)$ to a subtrace of type `TRACE Bool` by induction hypothesis. Therefore it has type `VALUE (TRACE Bool)` which is equal to `Bool` by Lemma 37.
  For all base types D, $\mathtt{If}\ \langle \mathtt{cond} = [\![L]\!], \mathtt{out} = \mathbb{H}\rangle$ has type `Trace` D assuming $\mathbb{H} : \mathtt{Trace}\ D$. We have $[\![M]\!] : \mathsf{T}(\mathsf{TRACE}\ C)$ by IH. Therefore, by Lemma 10, the whole term obtained by *dist* has type `TRACE` C. The else-branch is analogous and the whole expression has type $\mathsf{T}(\mathsf{TRACE}\ C)$.
- Case $\dfrac{}{\Gamma \vdash [] : \mathtt{List}\ A}$:
  $$\dfrac{\dfrac{\dfrac{[\![\Gamma]\!] \vdash \mathsf{T}(\mathsf{TRACE}\ C) : \mathit{Type}\ \text{using}\ A = \mathsf{T}(C)}{[\![\Gamma]\!] \vdash [] : \mathtt{List}\ \mathsf{T}(\mathsf{TRACE}\ C)}}{[\![\Gamma]\!] \vdash [] : \mathsf{T}(\mathtt{List}^*\ (\mathsf{TRACE}\ C))}}{[\![\Gamma]\!] \vdash [] : \mathsf{T}(\mathsf{TRACE}\ (\mathtt{List}^*\ C))}$$
- Case $\dfrac{\Gamma \vdash M : A}{\Gamma \vdash [M] : \mathtt{List}\ A}$:
  $$\dfrac{\dfrac{\dfrac{\text{IH}}{[\![\Gamma]\!] \vdash [\![M]\!] : \mathsf{T}(\mathsf{TRACE}\ C)}}{[\![\Gamma]\!] \vdash [[\![M]\!]] : \mathtt{List}\ \mathsf{T}(\mathsf{TRACE}\ C)}}{[\![\Gamma]\!] \vdash [[\![M]\!]] : \mathsf{T}(\mathsf{TRACE}\ (\mathtt{List}^*\ C))}$$
- Case $\dfrac{\Gamma \vdash M : \mathtt{List}\ A \qquad \Gamma \vdash N : \mathtt{List}\ A}{\Gamma \vdash M +\!\!+ N : \mathtt{List}\ A}$:
  $$\dfrac{\dfrac{\dfrac{\dfrac{\text{IH}}{[\![\Gamma]\!] \vdash [\![M]\!] : \mathsf{T}(\mathsf{TRACE}\ (\mathtt{List}^*\ C))}}{[\![\Gamma]\!] \vdash [\![M]\!] : \mathtt{List}\ \mathsf{T}(\mathsf{TRACE}\ C)} \qquad \text{analogous for } N}{[\![\Gamma]\!] \vdash [\![M]\!] +\!\!+ [\![N]\!] : \mathtt{List}\ \mathsf{T}(\mathsf{TRACE}\ C)}}{[\![\Gamma]\!] \vdash [\![M]\!] +\!\!+ [\![N]\!] : \mathsf{T}(\mathsf{TRACE}\ (\mathtt{List}^*\ C))}$$
- Case $\dfrac{\Gamma \vdash M : \mathtt{List}\ B \qquad \Gamma, x : B \vdash N : \mathtt{List}\ A}{\Gamma \vdash \mathbf{for}\ (x \leftarrow M)\ N : \mathtt{List}\ A}$:
  $$\dfrac{\dfrac{\dfrac{\dfrac{\text{IH}}{[\![\Gamma]\!] \vdash [\![M]\!] : \mathsf{T}(\mathsf{TRACE}\ (\mathtt{List}^*\ D))}}{[\![\Gamma]\!] \vdash [\![M]\!] : \mathtt{List}\ \mathsf{T}(\mathsf{TRACE}\ D)} \qquad \dfrac{\star}{[\![\Gamma]\!], x : \mathsf{T}(\mathsf{TRACE}\ D) \vdash b : \mathtt{List}\ \mathsf{T}(\mathsf{TRACE}\ C)}}{[\![\Gamma]\!] \vdash \mathbf{for}\ (x \leftarrow [\![M]\!])\ b : \mathtt{List}\ \mathsf{T}(\mathsf{TRACE}\ C)}}{[\![\Gamma]\!] \vdash \mathbf{for}\ (x \leftarrow [\![M]\!])\ b : \mathsf{T}(\mathsf{TRACE}\ (\mathtt{List}^*\ C))}$$
  where $b = \mathit{dist}(\mathsf{TRACE}\ C, \mathtt{For}\ D\ \langle \mathtt{in} = x, \mathtt{out} = \mathbb{H}\rangle, [\![N]\!])$ and $\star$ follows from the induction hypothesis applied to $[\![N]\!]$ and Lemma 10.

- The case for records is similar to that for list concatenation, in that we have multiple subtraces where the induction hypothesis applies, we just collect them into a record instead of another list concatenation.
- Case record projection: The projection was well-typed before tracing, so the record term M contains label l with some type A. By induction hypothesis and $A = \mathtt{T(TRACE}\ C\mathtt{)}$ the trace of M contains label l with type $\mathtt{TRACE}\ C$.

$$\dfrac{\dfrac{\text{IH}}{[\![\Gamma]\!] \vdash [\![M]\!] : \langle l^\bullet : \mathtt{T(TRACE}\ C\mathtt{)}, \ldots\rangle}}{[\![\Gamma]\!] \vdash [\![M]\!].l : \mathtt{T(TRACE}\ C\mathtt{)}}$$

- Case **`table`**: This is a slightly more complicated version of the base case for constants. We essentially map the `Cell` trace constructor over every table cell. Thus we go from a list of records of base types to a list of records of `Traced` base types.

$$\dfrac{\dfrac{[\![\Gamma]\!] \vdash \mathbf{table}\ \ldots \qquad \dfrac{\dfrac{[\![\Gamma]\!], y : \langle \overline{l : C}\rangle \vdash y.l : C}{\star}}{[\![\Gamma]\!], y : \langle \overline{l : C}\rangle \vdash [\langle \overline{l = \mathit{cell}(n, l, y.\mathtt{oid}, y.l)}\rangle] : [\langle \overline{l : \mathtt{Trace}\ C}\rangle]}}{[\![\Gamma]\!] \vdash \mathbf{for}\ (y \leftarrow \mathbf{table}\ n\ \langle \overline{l : C}\rangle)\ [\langle \overline{l = \mathit{cell}(n, l, y.\mathtt{oid}, y.l)}\rangle] : [\langle \overline{l : \mathtt{Trace}\ C}\rangle]}}{[\![\Gamma]\!] \vdash \mathbf{for}\ (y \leftarrow \mathbf{table}\ n\ \langle \overline{l : C}\rangle)\ [\langle \overline{l = \mathit{cell}(n, l, y.\mathtt{oid}, y.l)}\rangle] : \mathtt{T(TRACE}\ [\langle \overline{l : C}\rangle]\mathtt{)}}$$

  There are a couple of steps missing at $\star$. The singleton list step is trivial. Then we have one precondition for each column in the table. Recall that *cell* is essentially an abbreviation for `Cell`, which records table name, column name, row number, and the actual cell data in a trace. We use the table name n and the record label l as string values for the table and column fields. We enforce in the typing rules that every table has the `oid` column of type `Int`.
- Case equality:

$$\dfrac{\dfrac{}{[\![\Gamma]\!] \vdash C : \mathit{Type}} \qquad \dfrac{\text{IH}}{[\![\Gamma]\!] \vdash [\![M]\!] : \mathtt{T(TRACE}\ C\mathtt{)}} \qquad \dfrac{\text{IH}}{[\![\Gamma]\!] \vdash [\![N]\!] : \mathtt{T(TRACE}\ C\mathtt{)}}}{[\![\Gamma]\!] \vdash \mathtt{OpEq}\ C\ \langle \mathtt{left} = [\![M]\!], \mathtt{right} = [\![N]\!]\rangle : \mathtt{Trace\ Bool}}$$

- Case plus, with liberal application of `T(TRACE Int) = Trace Int`:

$$\dfrac{\dfrac{\text{Induction hypothesis}}{[\![\Gamma]\!] \vdash [\![M]\!] : \mathtt{T(TRACE\ Int)}} \qquad \dfrac{\text{Induction hypothesis}}{[\![\Gamma]\!] \vdash [\![N]\!] : \mathtt{T(TRACE\ Int)}}}{[\![\Gamma]\!] \vdash \mathtt{OpPlus}\ \langle \mathtt{left} = [\![M]\!], \mathtt{right} = [\![N]\!]\rangle : \mathtt{T(TRACE\ Int)}}$$

□

C.6 Proof of Lemma 13

Proof. By induction on the kinding derivation. We look at the possible reductions (see Figure 26). Congruence rules allow for reduction in rows, function bodies, applications, list, trace, record, row map, and typerec. These all follow directly from the induction hypothesis. The remaining cases are:

- $(\lambda\alpha : K.C)\ D \rightsquigarrow C[\alpha := D]$: by Lemma 34.
- **`Rmap`** $C\ \cdot \rightsquigarrow \cdot$: both sides have kind *Row*.
- **`Rmap`** $C\ (l : D; S) \rightsquigarrow (l : C\ D; \mathbf{Rmap}\ C\ S)$: from the induction hypothesis we have that C has kind $\mathit{Type} \to \mathit{Type}$, D has kind *Type*, and S has kind *Row*. Therefore $C\ D$ has kind *Type* and the whole right-hand side has kind *Row*.
- Typerec β-rules:
  - Base type right hand sides have kind *Type* by IH.
  - Lists:

$$\mathbf{Typerec}\ \mathtt{List}^*\ D\ (C_B, C_I, C_S, C_L, C_R, C_T) \rightsquigarrow C_L\ D\ (\mathbf{Typerec}\ D\ (C_B, C_I, C_S, C_L, C_R, C_T))$$

    C_L has kind $\mathit{Type} \to K \to K$ by IH. D has kind *Type* by IH, and the typerec expression has kind K.
  - Records:

$$\mathbf{Typerec}\ \mathtt{Record}^*\ S\ (C_B, C_I, C_S, C_L, C_R, C_T) \rightsquigarrow C_R\ S\ (\mathbf{Rmap}\ (\lambda\alpha.\mathbf{Typerec}\ \alpha\ (C_B, C_I, C_S, C_L, C_R, C_T))\ S)$$

    C_L has kind $\mathit{Row} \to \mathit{Row} \to K$ by IH. S has kind *Row* by IH. The row map expression has kind *Row*, because the type-level function has kind $\mathit{Type} \to \mathit{Type}$.
  - The trace case is analogous to the list case. □

C.7 Proof of Lemma 14

Proof. By induction on the typing derivation $\Gamma \vdash M : A$. Constants, variables, empty lists, and empty records do not reduce. We omit discussion of the cases that follow directly from the induction hypothesis, Lemma 13, and congruence rules (see Figure 33), like $M + N$ being able to reduce in both M and N. The remaining, interesting reduction rules are the β-rules in Figure 31 and the commuting conversions in Figure 32. We discuss them grouped by the relevant typing rule.

- Function application:
  - $(\lambda x.M)\ N \rightsquigarrow M[x := N]$: follows from Lemma 34.
  - $(\mathbf{if}\ L\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2)\ N \rightsquigarrow \mathbf{if}\ L\ \mathbf{then}\ M_1\ N\ \mathbf{else}\ M_2\ N$:
    We have:
    $$\dfrac{\dfrac{\Gamma \vdash L : \mathtt{Bool} \qquad \Gamma \vdash M_1 : A \to B \qquad \Gamma \vdash M_2 : A \to B}{\Gamma \vdash \mathbf{if}\ L\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : A \to B} \qquad \Gamma \vdash N : A}{\Gamma \vdash (\mathbf{if}\ L\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2)\ N : B}$$
    and can therefore show:
    $$\dfrac{\Gamma \vdash L : \mathtt{Bool} \qquad \dfrac{\Gamma \vdash M_1 : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M_1\ N : B} \qquad \dfrac{\Gamma \vdash M_2 : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M_2\ N : B}}{\Gamma \vdash \mathbf{if}\ L\ \mathbf{then}\ M_1\ N\ \mathbf{else}\ M_2\ N : B}$$
- Type instantiation:
  - $(\Lambda\alpha.M)\ C \rightsquigarrow M[\alpha := C]$: follows from the constructor substitution lemma (Lemma 34).
  - $(\mathbf{if}\ L\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2)\ C \rightsquigarrow \mathbf{if}\ L\ \mathbf{then}\ M_1\ C\ \mathbf{else}\ M_2\ C$: hoisting if-then-else out of the term works the same as application above.
- Fixpoint: follows from the substitution lemma (Lemma 34).
- If-then-else: if the condition is a Boolean constant, the expression reduces to the appropriate branch, which has the correct type by IH. The commuting conversion for lifting if-then-else out of the condition is type-correct by IH and rearranging of if-then-else rules.
- List comprehensions:
  - The if-then-else commuting conversion is as before.
  - $\mathbf{for}\ (x \leftarrow [])\ N \rightsquigarrow []$: [] has any list type and N has a list type.
  - $\mathbf{for}\ (x \leftarrow [M])\ N \rightsquigarrow N[x := M]$: by substitution (Lemma 34).
  - $\mathbf{for}\ (x \leftarrow M_1 +\!\!+ M_2)\ N \rightsquigarrow (\mathbf{for}\ (x \leftarrow M_1)\ N) +\!\!+ (\mathbf{for}\ (x \leftarrow M_2)\ N)$: reorder rules.
  - $\mathbf{for}\ (x \leftarrow \mathbf{for}\ (y \leftarrow L)\ M)\ N \rightsquigarrow \mathbf{for}\ (y \leftarrow L)\ \mathbf{for}\ (x \leftarrow M)\ N$:
    We have:
    $$\dfrac{\dfrac{\Gamma \vdash L : [A_L] \qquad \Gamma, y : A_L \vdash M : [A_M]}{\Gamma \vdash \mathbf{for}\ (y \leftarrow L)\ M : [A_M]} \qquad \Gamma, x : A_M \vdash N : [A_N]}{\Gamma \vdash \mathbf{for}\ (x \leftarrow \mathbf{for}\ (y \leftarrow L)\ M)\ N : [A_N]}$$
    We need:
    $$\dfrac{\Gamma \vdash L : [A_L] \qquad \dfrac{\Gamma, y : A_L \vdash M : [A_M] \qquad \Gamma, y : A_L, x : A_M \vdash N : [A_N]}{\Gamma, y : A_L \vdash \mathbf{for}\ (x \leftarrow M)\ N : [A_N]}}{\Gamma \vdash \mathbf{for}\ (y \leftarrow L)\ \mathbf{for}\ (x \leftarrow M)\ N : [A_N]}$$
    We obtain $\Gamma, y : A_L, x : A_M \vdash N : [A_N]$ from $\Gamma, x : A_M \vdash N : [A_N]$ by weakening (Lemma 35) and context swap (Lemma 36).
- Projection: The β rule is obvious, the if-then-else commuting conversion is as before.
- Type equality $\dfrac{\Gamma \vdash N : B \qquad \Gamma \vdash A = B}{\Gamma \vdash N : A}$: for all N' with $N \rightsquigarrow N'$ we have that $\Gamma \vdash N' : B$ by the induction hypothesis.
  We also know that $\Gamma \vdash A = B$, so $\Gamma \vdash N' : A$ by this typing rule and symmetry of type equality.
- Case **rmap**: Typing rule:
  $$\dfrac{\Gamma \vdash M : \forall\alpha : \mathit{Type}.\mathsf{T}(\alpha) \to \mathsf{T}(C\ \alpha) \qquad \Gamma \vdash N : \mathsf{T}(\mathsf{Record}^*\ S)}{\Gamma \vdash \mathbf{rmap}^S\ M\ N : \mathsf{T}(\mathsf{Record}^*\ (\mathbf{Rmap}\ C\ S))}$$
  Reduction rule:
  $$\mathbf{rmap}^{\overline{\langle l_i : C_i \rangle}}\ M\ N \rightsquigarrow \overline{\langle l_i = (M\ C_i)\ N.l_i \rangle}$$

Need to show that $\langle \overline{l_i = (M\ C_i)\ N.l_i} \rangle : \mathsf{T}(\mathsf{Record}^*\ (\mathbf{Rmap}\ C\ \langle \overline{l_i : C_i} \rangle))$. By row type constructor evaluation, that type equals $\mathsf{T}(\mathsf{Record}^*\ \langle \overline{l_i : C\ C_i} \rangle)$, which is the obvious type of $\langle \overline{l_i = (M\ C_i)\ N.l_i} \rangle$.

- Case **rfold**: Typing rule:

$$\frac{\Gamma \vdash L : \mathsf{T}(C) \to \mathsf{T}(C) \to \mathsf{T}(C) \qquad \Gamma \vdash M : \mathsf{T}(C) \qquad \Gamma \vdash N : \mathsf{T}(\mathsf{Record}^*\ (\mathbf{Rmap}\ (\lambda\alpha.\alpha \to C)\ S))}{\Gamma \vdash \mathbf{rfold}^S\ L\ M\ N : \mathsf{T}(C)}$$

Reduction rule:

$$\mathbf{rfold}^{(\overline{l_i:C_i})}\ L\ M\ N \rightsquigarrow L\ N.l_1\ (L\ N.l_2 \ldots (L\ N.l_n\ M) \ldots)$$

Need to show that $L\ N.l_1\ (L\ N.l_2 \ldots (L\ N.l_n\ M) \ldots)$ has type $\mathsf{T}(C)$. M has type $\mathsf{T}(C)$. L has type $\mathsf{T}(C) \to \mathsf{T}(C) \to \mathsf{T}(C)$. Each $N.l_i$ has type $\mathsf{T}(C)$, because N has a record type obtained by mapping the constant function with result C over row S.

- Typecase typing rule:

$$\frac{\begin{array}{c}\Gamma \vdash C : \mathit{Type} \qquad \Gamma, \alpha : \mathit{Type} \vdash B : \mathit{Type} \\ \beta, \rho, \gamma \notin \mathit{Dom}(\Gamma) \qquad \Gamma \vdash M_B : B[\alpha := \mathsf{Bool}^*] \qquad \Gamma \vdash M_I : B[\alpha := \mathsf{Int}^*] \qquad \Gamma \vdash M_S : B[\alpha := \mathsf{String}^*] \\ \Gamma, \beta : \mathit{Type} \vdash M_L : B[\alpha := \mathsf{List}^*\ \beta] \qquad \Gamma, \rho : \mathit{Row} \vdash M_R : B[\alpha := \mathsf{Record}^*\ \rho] \qquad \Gamma, \gamma : \mathit{Type} \vdash M_T : B[\alpha := \mathsf{Trace}^*\ \gamma]\end{array}}{\Gamma \vdash \mathbf{typecase}^{\alpha.B}\ C\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) : B[\alpha := C]}$$

Reduction rules:

– **typecase** Bool^* **of** $(M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) \rightsquigarrow M_B$
Need to show that $M_B : B[\alpha := \mathsf{Bool}^*]$, which is one of our hypotheses.

– **typecase** $\mathsf{List}^*\ C$ **of** $(M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) \rightsquigarrow M_L[\beta := C]$
Need to show that the result of reduction $M_L[\beta := C]$ has type $B[\alpha := \mathsf{List}^*\ C]$, the same as the typing rule.

$$\frac{}{\Gamma \vdash M_L[\beta := C] : B[\alpha := \mathsf{List}^*\ C]}$$

Instantiating the constructor substitution lemma (Lemma 34) gives us

$$\Gamma[\beta := C] \vdash M_L[\beta := C] : (B[\alpha := \mathsf{List}\ \beta])[\beta := C]$$

from $\Gamma, \alpha : \mathit{Type} \vdash B : \mathit{Type}$ and $\beta \notin \mathit{Dom}(\Gamma)$ we know that neither B nor Γ can contain β. Thus the only substitution for β we need to perform is in the substitution for α and we can reassociate substitution like this:

$$\Gamma \vdash M_L[\beta := C] : B([\alpha := \mathsf{List}\ \beta][\beta := C])$$

which is the same as

$$\Gamma \vdash M_L[\beta := C] : B[\alpha := \mathsf{List}\ C]$$

The other cases are analogous.

- Case **tracecase**: Typing rule:

$$\frac{\begin{array}{c}\Gamma \vdash M : \mathsf{Trace}\ A \qquad \Gamma, x_L : A \vdash M_L : B \qquad \Gamma, x_I : \langle \mathsf{cond} : \mathsf{Trace\ Bool}, \mathsf{then} : \mathsf{Trace}\ A \rangle \vdash M_I : B \\ \Gamma, \alpha_F : \mathit{Type}, x_F : \langle \mathsf{in} : \mathsf{T}(\mathsf{TRACE}\ \alpha_F), \mathsf{out} : \mathsf{Trace}\ A \rangle \vdash M_F : B \qquad \Gamma, x_C : \langle \mathsf{table} : \mathsf{String}, \mathsf{column} : \mathsf{String}, \mathsf{row} : \mathsf{Int}, \mathsf{data} : A \rangle \vdash M_C : B \\ \Gamma, \alpha_E : \mathit{Type}, x_E : \langle \mathsf{left} : \mathsf{T}(\mathsf{TRACE}\ \alpha_E), \mathsf{right} : \mathsf{T}(\mathsf{TRACE}\ \alpha_E) \rangle \vdash M_E : B \qquad \Gamma, x_P : \langle \mathsf{left} : \mathsf{Trace\ Int}, \mathsf{right} : \mathsf{Trace\ Int} \rangle \vdash M_P : B\end{array}}{\Gamma \vdash \mathbf{tracecase}\ M\ \mathbf{of}\ (x_L.M_L, x_I.M_I, \alpha_F.x_F.M_F, x_C.M_C, \alpha_E.x_E.M_E, x_P.M_P) : B}$$

Reductions:

– **tracecase** $\mathsf{For}\ C\ M$ **of** $(x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P) \rightsquigarrow M_F[\alpha := C, x := M]$
We need to show $\dfrac{\star}{\Gamma \vdash M_F[\alpha := C, x := M] : A}$
$\star$: We only need $M : \langle \mathsf{in} : \ldots \rangle$ and $C : \mathit{Type}$, which we get by inversion of the typing rule for For and the substitution lemmas.

The other cases are analogous. □

C.8 Proof of Lemma 17

Proof. By induction on the kinding derivation of C or S (see Figure 24).

- Base types `Bool`, `Int`, `String` are in normal form.
- Type variables α are in normal form.
- Type-level functions $\lambda\alpha.C$: by IH, either $C \rightsquigarrow C'$, in which case $\lambda\alpha.C \rightsquigarrow \lambda\alpha.C'$, or C is in normal form already, in which case $\lambda\alpha.C$ is in normal form, too.
- Type-level application $C\ D$: by IH either C or D may reduce, in which case the whole application reduces. Otherwise, C and D are in normal form. The following cases of C do not apply, because they are ill-kinded: base types, lists, records, and traces. If C is a normal form and a variable, application, or typerec then C is a neutral form and D is a normal form so $C\ D$ is a neutral (and normal) form. Finally, if C is a type-level function, the application β-reduces.
- List types: by IH either the argument reduces, or is in normal form already.
- Record types: by IH either the argument (a row) reduces, or is in normal form already.
- Trace types: by IH either the argument reduces, or is in normal form already.
- **Typerec** C **of**$(C_B, C_I, C_S, \alpha.C_L, \rho.C_R, \alpha.C_T)$: by IH, either $C \rightsquigarrow C'$, in which case **Typerec** reduces with a congruence rule, or C is in one of the following normal forms:
  - If C is a base, list, record, or trace constructor, the **Typerec** expression β-reduces to the respective branch.
  - C cannot be a type-level function, that would be ill-kinded.
  - If C is one of the following neutral forms: variables, applications, and **Typerec**, then by IH the branches C_B, C_I, etc. either reduce and a congruence rule applies, or they are all in normal form and **Typerec** C **of**$(C_B, C_I, C_S, \alpha.C_L, \rho.C_R, \alpha.C_T)$ is in normal form.
- The empty row $\cdot$ is in normal form.
- Row extensions $l : C; S$: by IH applied to C and S we have three cases:
  - If $C \rightsquigarrow C'$, then $l : C; S \rightsquigarrow l : C'; S$.
  - If $S \rightsquigarrow S'$, then $l : C; S \rightsquigarrow l : C; S'$.
  - If C and S are in normal form, then $l : C; S$ is in normal form.
- **Rmap** $C\ S$: we apply the induction hypothesis to S and C. If either C or S takes a step, the whole row map expression takes a step via the respective congruence rule. Otherwise S is in one of the following normal forms:
  - Case empty row: **Rmap** $C\ \cdot \rightsquigarrow \cdot$
  - Case $l : D; S'$: **Rmap** $C\ (l : D; S') \rightsquigarrow (l : C\ D;$ **Rmap** $C\ S')$.
  - Case **Rmap** $D\ U$: **Rmap** C (**Rmap** $D\ U$) is in normal form.
  - Case ρ: **Rmap** $C\ \rho$ is in normal form.
- The row variable ρ is in normal form. □

C.9 Proof of Lemma 18

Proof. By induction on the typing derivation of M.

- Constants: in normal form.
- Term variables: in normal form.
- Term function: apply IH to body and either reduce or in normal form.
- Fixpoint: we can always take a step by unrolling once.
- Term application $M\ N$: apply induction hypothesis to M. If M reduces to M', then $M\ N$ reduces to $M'\ N$. Otherwise, M is in Links$^{\mathrm{T}}$ normal form. It cannot be any of the following, because these would be ill-typed: constants, type abstraction, operators, record introduction forms including record map, list introduction forms, trace introduction forms. In the following cases, we apply the induction hypothesis to N and either reduce to $M\ N'$ or are in normal form already: variable, application, type application, record fold, tracecase, typecase. This leaves the following cases:
  - If M is a function, we β-reduce.
  - If M is of the form if-then-else, we reduce using a commuting conversion.
- Term-level type abstraction $\forall\alpha : M$: by IH, either $M \rightsquigarrow M'$, in which case $\forall\alpha : M \rightsquigarrow \forall\alpha : M'$, or M is in normal form, in which case $\forall\alpha : M$ is in normal form as well.
- Term-level type application $M\ C$: apply induction hypothesis to M. If M reduces to M', then $M\ C$ reduces to $M'\ C$. Otherwise, M is in Links$^{\mathrm{T}}$ normal form. It cannot be any of the following, because these would be ill-typed: constants, functions, operators, record introduction forms including record map, list introduction forms, trace introduction forms. In the following cases, the application is already in normal form: variable, application, type application, projection, record fold, tracecase, typecase. This leaves the following cases:

  - If it is a term-level type abstraction, we β-reduce.
  - If it is of the form if-then-else, we perform a commuting conversion.
- Case **if** L **then** M **else** N: apply induction hypothesis to all subterms. If any of the subterms reduce, then the whole if-then-else reduces. Otherwise, L, M, N are in Links$^{\mathrm{T}}$ normal form. The condition cannot be any of the following, because these would be ill-typed: functions, type abstractions, arithmetic operators, record introduction forms including record map, list introduction forms, trace introduction forms. In the following cases, the condition already matches the normal form: variable, application, type application, projection, record fold, tracecase, and typecase. This leaves the following cases for the condition:
  - Constants: **true** and **false** reduce, other constants are ill-typed.
  - If the condition is of the form if-then-else itself, we apply a commuting conversion.
  - Operators with Boolean result like == are in normal form.
- Records $\langle l = M; N\rangle$: apply induction hypothesis to M and N. If either reduces, the whole record reduces, otherwise it is in normal form.
- Projection $M.l$: apply induction hypothesis to M. If M reduces to M', then $M.l$ reduces to $M'.l$. Otherwise, M is in Links$^{\mathrm{T}}$ normal form. It cannot be any of the following, because these would be ill-typed: constants, functions, type abstraction, operators, list introduction forms, trace introduction forms. In any of the following cases of M, $M.l$ is already in normal form: variable, application, type application, projection, record map, record fold, typecase, tracecase. This leaves the following cases for M:
  - If it is of the form if-then-else itself, we apply a commuting conversion.
  - It cannot be an empty record, or a record expression where label l does not appear—these would be ill-typed. If M is a record literal that maps l to M' then $\langle l = M'; N\rangle.l$ reduces to M'.
- Record map $\mathbf{rmap}^S\ M\ N$: by Lemma 17 we have that either S reduces to S', in which case $\mathbf{rmap}^S\ M\ N$ reduces to $\mathbf{rmap}^{S'}\ M\ N$, or is in normal form. Similarly, M and N may reduce by IH. Otherwise, we have S, M, and N in normal form. By cases of S:
  - If it is a closed row, we apply the β-rule.
  - If it is an open row U, $\mathbf{rmap}^U\ M\ N$ is in normal form.
- Record fold $\mathbf{rfold}^S\ L\ M\ N$: same as record map.
- Empty list: in normal form.
- Singleton list: apply IH to element and reduce or is in normal form.
- List concatenation: apply IH to both sides. If either reduces, the whole concatenation reduces, otherwise it is in normal form.
- Comprehension **for** $(x \leftarrow M)\ N$: apply induction hypothesis to M. If M reduces to M' then **for** $(x \leftarrow M)\ N$ reduces to **for** $(x \leftarrow M')\ N$. Otherwise, M is in Links$^{\mathrm{T}}$ normal form. It cannot be any of the following, because these would be ill-typed: constants, functions, type abstractions, primitive operators, record introduction forms including record map, and trace constructors. In the following cases we apply the IH to the body and either reduce or the whole comprehension is in normal form: variables, term application, type application, projection, tables, record fold, tracecase, typecase. This leaves the following cases for M:
  - If-then-else: reduces with a commuting conversion.
  - Empty list: the whole comprehension reduces to the empty list.
  - Singleton list: β-reduces.
  - List concatenation: reduces with a commuting conversion.
  - Comprehension: reduces with a commuting conversion.
- Table: in normal form.
- Trace constructors: apply IH and Lemma 17 to constituent parts. If either reduces, the whole trace constructor reduces, otherwise it is in normal form.
- Tracecase: apply induction hypothesis to the scrutinee. If it reduces, the whole tracecase expression reduces. Otherwise it is in Links$^{\mathrm{T}}$ normal form. It cannot be any of the following, because these would be ill-typed: constants, functions, type abstractions, primitive operators, record introduction forms, record map, empty or singleton lists, list concatenations or comprehensions, tables. If the scrutinee is any of the following, by IH we reduce in the branches or the whole tracecase is in normal form: variables, term application, type application, projection, record fold, tracecase, typecase. This leaves the following cases:
  - If-then-else: reduces using commuting conversion.
  - Trace constructor: β-reduces.

- Typecase: apply Lemma 17 to the scrutinee. Either it reduces, in which case the whole typecase expression reduces. Otherwise it is in normal form. It cannot be a type-level function, that would be ill-kinded. In the following cases, we apply the induction hypothesis to the branches of the typecase and reduce there, or we are in Links$^{\mathrm{T}}$ normal form: type variables, type-level application, and typerec. And finally, if the outmost constructor is one of the following, a β-rule applies: bool, int, string, list, record, trace.
- Primitive operators like == and +: by IH either the arguments reduce, in which case the whole expression reduces, or are in normal form, in which case the whole expression is in normal form. □

C.10 Proof of Lemma 20

Proof. By induction on the typing derivation. The term cannot be a record fold or typecase, because those necessarily contain a (row) type variable, which is unbound in the query context Γ. It cannot be a term application, type application, or tracecase, because the term in function position or the scrutinee, by IH, is of the form x or $x.l$, both of which are ill-typed given that the query context Γ does not contain function types, polymorphic types, or trace types. Projections $P.l$ are of the form $F.l$ or $(\mathbf{rmap}^U\ M\ N).l$. The former case reduces by IH to $x.l$ or $x.l'.l$, the first of which is okay, and the second is ill-typed. The latter case is impossible, because U necessarily contains a row variable and would therefore be ill-typed. This leaves variables x and projections of variables $x.l$. □

C.11 Proof of Theorem 22

Proof. By induction on the typing derivation.

- Constants, variables, empty lists, and tables are in both languages.
- Functions, type abstractions, and trace constructors do not have nested relational type.
- Function application: The typing rule
$$\frac{\Gamma \vdash M' : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M'N : B}$$
requires M' to have a function type. Since M is in normal form, M' matches the grammar F. Lemma 20 implies that M' is either a variable x or a projection $x.l$. The query context Γ assigns record types with labels of base types to all variables — not function types — a contradiction.
- Type instantiation: The typing rule
$$\frac{\Gamma \vdash M' : \forall \alpha : K.A \qquad \Gamma \vdash C : K}{\Gamma \vdash M'\ C : A[\alpha := C]}$$
requires M' to have a polymorphic type. The normal form assumption requires M' to match the normal form F. Therefore, Lemma 20 applies, so M' is either a variable x or a projection $x.l$. The query context Γ assigns record types with labels of base types to all variables — a contradiction.
- Primitive operators, if-then-else, records, singleton list, and list concatenation: apply the induction hypothesis to the subterms.
- Projection $M'.l$: M' is in normal form P, which is either of the form F or a record map. Lemma 20 restricts F to x and $x.l'$, both of which are nested relational calculus terms. P cannot be of the form $\mathbf{rmap}^U\ N'\ N''$, because U necessarily contains a free type variable (see Remark 16), and thus cannot be well-typed in a query context Γ which does not contain type variables.
- Record map and fold have normal forms $\mathbf{rmap}^U\ M'\ N$ and $\mathbf{rfold}^U\ L\ M'\ N$, respectively. U necessarily contains a free type variable (see Remark 16), and thus cannot be well-typed in a query context Γ which does not contain type variables.
- List comprehension $\mathbf{for}\ (x \leftarrow M')\ N$: The iteratee M' is in normal form T, which includes tables and normal forms F. If M' is a table, x has closed record type with labels of base types, the induction hypothesis applies to N, and the whole expression is in nested relational calculus. If M' is of the form F, Lemma 20 applies and implies that M' is either x or $x.l$. Both cases are ill-typed, because the query context Γ only contains variables with closed records with labels of base type — a contradiction.
- Tracecase: much like the application case above, the typing derivation forces the scrutinee to be of trace type. The normal form forces the scrutinee to be of the form F, and from Lemma 20 follows that it has to be a variable, or projection of a variable. The query context Γ assigns record types with labels of base types to all variables — a contradiction.
- Typecase: the scrutinee is in normal form E which contains at least one free type variable (see Remark 16). In a query context which only binds term variables, this cannot possibly be well-typed — a contradiction. □