Language-integrated Provenance

Stefan Fehrenbach

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2019

Abstract

Provenance is metadata about the where, the why, and the how of data. It is evidence which can answer questions such as: Where exactly did this piece of data come from? Why is this row in my result? How was it produced? Answers to these questions are useful for judging the trustworthiness of data, and for finding and correcting mistakes.

Most programs that use a database at all, already use one crude form of provenance: they manually propagate row identifiers together with database values, just in case they need to be updated later. More sophisticated forms of provenance are exceedingly rare, because they are more difficult to implement manually. Tools to calculate data provenance systematically, only exist as research prototypes. Even standard database systems are hard to set up, as evidenced by the rise of hosted database services, so there is little suprise that prototypes of provenance systems are not used much.

This dissertation shows how a programming language can provide support for provenance. Based on language-integrated query technology, it can systematically rewrite queries to produce various forms of provenance. We describe such query transformations for where-provenance and lineage, and discuss how to enable programmers to define their own forms of provenance. Thanks to query normalization the resulting queries still execute efficiently on mainstream database systems. A programming language can help further by giving provenance metadata precise types to ensure that it is handled appropriately.

Language-integrated queries make it easy to write programs that deal with data, no special query language needed. Language-integrated provenance makes it as easy to deal with data provenance, no special database needed.

Lay Summary

Databases store the world's data. They provide us with answers to increasingly difficult questions. Sometimes the answers are wrong. Imagine a database of train connections. We could ask for a train from here to there via somewhere in particular, but only those that leave us enough time for a swift half at each station we need to change at. As we leave the train at somewhere, half-way between here and there, we realize that there is not actually enough time. The next train leaves in three minutes. The database was wrong about the departure time. To fix it, we need to figure out *where* in the database the departure time is stored. Similarly, we could end up stranded somewhere, with no more trains for the day, because the database thought a train existed, that does not. To fix this, we need to figure out *why* the database produced this connection.

Data provenance describes where data came from, why it exists, who modified it, how it was produced in the first place, and so forth. It can be used to answer the questions above, to find wrong data in databases, and fix it.

Unfortunately, most databases just give answers, not answers to questions about their answers — provenance. This dissertation shows we do not actually need the database to do anything special. When we write programs to retrieve data from a database, we can have the programming language automatically rewrite those programs to also compute data provenance. Essentially, instead of asking for a result, we can modify our question slightly to ask for the result together with its provenance. Any database that can answer the original question can also answer the modified question.

Hopefully, making access to data provenance independent of the database will make it more easily available. Then we can hope that in the future, it is easy to correct wrong data in a database, complicated computations will be transparent, and data can be trusted because its provenance is easily verified.

Acknowledgements

My thanks, first and foremost, to my advisor James Cheney for his generous guidance, encouragement, and unyielding support. I have left every single one of our meetings happier and more confident than I entered it, even and especially when things had gone poorly. I would also like to thank Peter Buneman, Ian Stark, James McKinna, and Jan Stolarek for their interest and feedback.

I would like to thank Torsten Grust and Sam Lindley for agreeing to be my examiners and their feedback, and Stephen Gilmore for chairing my viva.

Thanks to my awesome office mates Karoliina, Fabian, Ricardo, and Rudi; to the Friday lunch crowd Brian, Chad, Craig, Daniel, Frank, Garrett, Jack, Jakub, Simon, Weili, and Wen; and to the rest of the provenance group and the LFCS at large, for their companionship, distractions, discussions, laughs, and making the Forum such a welcoming and stimulating environment.

I would also like to thank my hosts and fellow interns at Microsoft Research and IBM, in particular Mike, Jérôme, Louis, and Avi, for a great time and everything I have learned there — even if not directly relevant to this dissertation. Thanks again also to James, for bringing these and other opportunities to my attention, enabling me to go, and for his patience when they took over my time.

I would not have ended up doing a PhD in Edinburgh without my parents Elvira & Klaus, my teachers Zwanger and Bitsch, and the Marburg group Klaus, Sebastian, Tillmann, Paolo, Christian, and Felix. Special thanks to Phil Wadler for forwarding my application to James — a most fortunate rejection.

I wish to thank friends and family, old and new, Papa, Oma, Tobi, Axel, Rebecca, Martin, and all the others for putting up with my lack of communication and infrequent visits. I will try to do better. For generous helpings of life in my work–life balance, I owe great thanks in particular to Edinburgh friends, flatmates, and Movie Time attendees Angharad, Dan, Daniel, Emilia, Hannah, JC, Jenny, Karo, Kieran, and Michael; to Neil of EUMS Chorus; and to Firbush.

Finally, Karoliina, thank you for your love and support in this and in life.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(S. Fehrenbach, 21 May 2019)

Table of Contents

Chapter 1

Introduction

Provenance[1] is "the origin or earliest known history of something" or "a record of ownership of a work of art or an antique, used as a guide to authenticity or quality" [Oxford Dictionary, 2018]. Data provenance is information about the origin and derivation of data. Since informatics is the study of information processing and provenance is information about information and how it was processed, it is only natural that provenance crops up everywhere in informatics, although frequently not under that name. For example, Make [Feldman, 1979], the build system, uses file modification time metadata and explicit dependency information between files to provide incremental builds. This is a crude approximation of what a build system really needs to know, namely the provenance of a build result, to be able to rebuild if and only if any of the inputs have changed. Programming languages usually report errors annotated with information about where they occurred in the source file. The version control system Git reads configuration parameters from so many potential sources that they recently added an option to display where any value actually came from to help with debugging configurations.[2] All of these are matters of provenance.

Relational databases are the area of informatics where provenance itself has received the most attention. Given a query and a database, data provenance is additional information that explains the query result. It is important for assessing the trustworthiness of data and can be useful for identifying and correcting mistakes. Data and query languages are constrained enough to give

[1] From the French *provenir* (meaning 'to come from' or 'to stem from'), from the Latin *provenire*, from *pro-* (meaning 'forth') and *venire* (meaning 'come') [Oxford Dictionary, 2018].

[2] `https://github.com/blog/2131-git-2-8-has-been-released` accessed on 30 January 2018, Git commit `dd0f567f1041a3caea7856b3efe20f8fb9b487b5`.

nth	name
1	George Washington
⋮	⋮
44	Barack Obama
45	Donald Trump

(a) **presidents**

nth	term	date
1	1	4/30/1789
⋮	⋮	⋮
44	1	1/20/2009
44	2	1/21/2013
45	1	1/20/2017

(b) **inaugurations**

date	time	trips
1/21/2013	11	317k
1/20/2009	11	513k
1/20/2005	11	197k
1/20/2017	11	193k
⋮	⋮	⋮

(c) **metro**

Figure 1.1: A database of presidents, their inaugurations, and Metro ridership.

precise definitions for provenance and study its properties formally. Buneman, Khanna, and Tan [2001] and other researches have identified several different forms of provenance that answer different questions one might have about query results. For example, *input-provenance* describes what inputs were used to produce a result; *where-provenance* describes where each part of the result was copied from in the input; *why-provenance* describes why a result exists at all.

Consider the database in Figure 1.1 which consists of a table of presidents of the United States of America; the dates of their inaugurations [Coleman, 2017]; and hourly cumulative ridership data published by the Washington D.C. Metro.[3] Since inauguration attendance is not officially monitored, we might want to use the cumulative trips at 11:00 on a particular day as a proxy for the size of an inauguration. Given this data and interpretation, we can write a query to return the names and inauguration dates of presidents whose inauguration was at least as well-attended as Trump's in 2017.

In a programming language with language-integrated queries, such as Links [Cooper et al., 2007], this could look like the program in Figure 1.2. We first define two helper functions. The first function, `inaugDatesByPres`, takes a record `p` representing a president as its argument and uses a for-comprehension to iterate over the inaugurations table, filtering inaugurations on column `nth` to match the president `p`, and returning the inauguration dates as a list. The second function, `over193000`, iterates over the dates obtained by calling the first function and the Metro data to find dates where the cumulative ridership (column `trips`) at 11:00 was greater than 193000 — the number of trips on

[3] `https://twitter.com/wmata/status/822482330346487810` with the following correction `https://twitter.com/wmata/status/822487600158142464`

```
fun inaugDatesByPres(p) {
  for (i <- inaugurations) where (i.nth == p.nth) [i.date] }

fun over193000(p) {
  for (date <- inaugDatesByPres(p)) for (m <- metro)
  where (m.date == date && m.time == 11 && m.trips >= 193000)
    [date] }

query { for (p <- presidents)
        where (not(empty(over193000(p))))
          [(president = p.name,
            dates = over193000(p))] }
```

Figure 1.2: A Links program to find presidents and their inaugurations with Metro ridership ≥ 193000.

Trump's inauguration date. The query itself iterates over the presidents and returns their names and inauguration dates with higher attendance than Trump's, if there are any. Given our source database, the result looks like Table 1.1.

president	dates
⋮	⋮
Barack Obama	[1/21/2013, 1/20/2009]
Donald Trump	[1/20/2017]

Table 1.1: Presidents and their inaugurations with Metro ridership ≥ 193000.

Unlike C#'s Linq [Meijer et al., 2006], which popularized language-integrated query, Links supports efficient nested relational queries [Cheney et al., 2014c]. In the above example, the `dates` column of the result is itself a collection of dates. The example also shows the use of functions to abstract over parts of a query. Invisible to the user, Links generates two flat Sql queries, sends them to the database for evaluation, and combines their results into the nested representation. Support for nested data and functional abstraction is important for the rest of this work. They are also examples of features not found in mainstream database systems that can be added by the programming language.[4]

[4] Arrays, even though standardized in Sql:1999, are not first-class citizens in Sql.

president	dates
⋮	⋮
Barack Obama$^{p:name:44}$	[1/21/2013, 1/20/2009]$^{p:44,i:65,i:66,m:1,m:2}$
Donald Trump$^{p:name:45}$	[1/20/2017]$^{p:45,i:67,m:4}$

Table 1.2: The same result as Table 1.1, but annotated with president's where-provenance and dates' lineage.

Now, imagine the result of the above query was displayed on a website and a user complained about the last row reading "Donald Drumpf". While slightly amusing in a juvenile way, we would surely want to correct the mistake in the database. There is a form of provenance called where-provenance that answers exactly the question of where in a source database a particular cell of a query result was copied from [Buneman et al., 2001]. (Admittedly, in this example the where-provenance of the first column is not exactly complicated and would just tell us to go look at the respective rows of the presidents table.) Another user might be angry with us because they did not expect more than one result at all, given that Sean Spicer said that Trump had "the largest audience ever to witness an inauguration"[5]. Lineage [Cui et al., 2000] is a form of why-provenance and it answers the question of why a result exists. For each row in a query result, lineage gives evidence for its existence in the form of rows in the input database. To give a satisfactory reply to our angry users, we need to investigate the provenance of the result, in particular the where-provenance and lineage.

A provenance system could provide us with an annotated result, such as the one in Table 1.2. Each name is annotated with a reference to a database cell, where p stands for the presidents table. Thus we can double check the presidents table, name column, row 45; find that it does indeed say "Donald Trump"; and suggest to our first unhappy user that they might want to disable their browser extensions.[6] Similarly, the inauguration dates in the second column are annotated with rows that were visited while producing the result. Barack Obama's inaugurations are listed because he appears in row 44 of the presidents

[5] `https://www.theguardian.com/us-news/2017/jan/22/trump-inauguration-crowd-sean-spicers-claims-versus-the-evidence`

[6] Yes, such browser extensions exist. People have even inadvertently copy-and-pasted passages from modified websites into other documents and published them. You may blame John Oliver, or insufficient attention to provenance, as you wish.

table, has two inaugurations listed in rows 65 and 66 of the inaugurations table, which match rows 1 and 2 in the metro table with more than 193000 trips. We can point our second unhappy user to this evidence. They might complain about our methodology and sources, or compare the evidence against Spicer's and realize that his "alternative facts" are just lies.

A number of provenance systems (see Section 2.1.2 for an overview) have been proposed that augment database systems with support for querying provenance to produce annotated results such as the above. Unfortunately, to this date, none of the proposed provenance extensions to the query language SQL have been integrated into the standard, let alone been implemented by mainstream database systems. Today, if a user wants to have their provenance questions answered, they need to either use a research prototype of a provenance system which they might well have to compile from source and patch themselves, use some middleware system that intercepts and rewrites queries, or install plugins to their existing database. There are users for whom the more invasive of these options may not be available at all, because they cannot make changes to the database they use. Other users may prefer a provenance solution that works with a mainstream database, even one hosted as a service, without needing to run additional software.

Furthermore, the question of how a client should interact with provenance data has not been addressed at all in the literature. Many programmers use libraries and frameworks for object-relational mapping (like HIBERNATE in JAVA or ACTIVERECORD in RUBY) or language-integrated query (like LINQ for C#) to generate SQL queries for them. There are good reasons for using them, but they also cause problems for provenance queries. None of these client abstractions provide any support for provenance queries. To query provenance and debug their data, a user would have to find out what queries are generated for them and change them, leaving the comfort of their abstraction behind. Even when provenance is used in the provenance system's native language, how the user interacts with it has so far been an afterthought. To achieve high performance, systems like PERM return provenance annotations in a variable number of additional columns, which depends on the shape of the query. In the presence of dynamic queries, it is difficult to expose this safely in a client library. Beyond that, we might well want to treat provenance metadata differently from data in the program. For instance, provenance can be used to encode secrecy levels

[Corcoran et al., 2009]. If we build access control restrictions based on provenance, we might want to use types to guarantee it is not forged or accidentally misattributed.

In their contribution to the Festschrift in honor of Peter Buneman — "one of the first to recognize the importance of data provenance" — Glavic, Miller, and Alonso [2013] identify "four requirements for relational provenance systems":

1. "Support for different types of provenance with sound semantics."

   Since different forms of provenance answer different questions, a provenance system should support multiple forms of provenance to be able to offer its user a more complete understanding of their data.

2. "Support for provenance generation for complex [queries]."

   The richer the query language, the better. Ideally a system can generate provenance for queries that use all of the language, however, there are some natural limits to this. For example, if a query adds two numbers, the result will just not have any meaningful where-provenance because the value was not copied from the database. Unfortunately, provenance systems sometimes impose additional restrictions.

3. "Support for complex queries over provenance information."

   Provenance is metadata, which means it is also still data itself, and a user might well expect to be able to use the full power of their query language to filter, manipulate, and analyze it.[7] In some provenance systems, the data models for provenance annotations and the underlying data are different and thus the same language cannot be used to analyze them.

4. "Support for large databases."

   Provenance can be large — larger than the original database even. It should not be necessary to compute and store provenance for the whole database to get annotations on one query. This is particularly interesting in combination with Requirement 3: if we use provenance in a filter condition, for example, the system should avoid generating provenance for results that will be filtered out.

[7] Provenance of provenance is an under-explored topic, so we are inclined to forgive systems for not supporting provenance queries over provenance queries, for now.

This dissertation proposes language-integrated provenance as a database-independent implementation strategy for provenance systems. The core idea is that a programming language itself can provide access to provenance by systematically rewriting queries to compute provenance. This builds on top of existing techniques for language-integrated queries and requires no further support from the database system. In a hypothetical language with provenance support, we could implement the scenario from above by requesting where-provenance of the first column and lineage of the second column using the made up syntax **`with where-prov`** and **`with lineage`** as seen in the query below.

```
query {
  for (p <- presidents)
  where (not(empty(over193000(p))))
    [(president = p.name with where-prov,
      dates = over193000(p) with lineage)] }
```

The special provenance features are implemented by translation. For example, a plain Links query that computes a similar result as the provenance query above, could look something like the following.[8]

```
query {
  for (p <- presidents) where (not(empty(over193000(p))))
    [(president = (p.name, ("p", "name", p.nth))
      dates = (over193000(p),
               [("p", p.nth)] ++
                 for (d <- for (i <- inaugurations)
                           where (i.nth == p.nth) [i.date])
                 for (m <- metro)
                 where (m.date == d && m.time == 11 &&
                        m.trips >= 193000)
                   [("i", i.oid), ("m", m.oid)]))] }
```

This is still not an Sql query and thus cannot be executed by a mainstream database system. However, it is a valid Links program and the **`query`** keyword makes Links compile it to Sql and execute it on the database. By building on Links, we can keep the translation of provenance features simple and compositional and rely on the language to eliminate incidental complexities. More importantly, we get to use features such as higher-order functions in queries and nested collections in query results. Note how the dates carry a list of lineage annotations. This representation is natural in Links but not in Sql, especially

[8] We use the special `oid` column in place of row numbers.

if you consider Requirements 3 and 4 from above. If we used SQL arrays to encode lineage annotations, we would need to use a different subset of the language to operate on annotations, namely the built-in functions for arrays rather than the usual `SELECT`, `FROM`, `WHERE`. Contrast this with LINKS, where lineage is just another nested collection that we can iterate over using **`for`**, filter with **`where`**, and combine and compare with other data at will. Furthermore, generated queries including nested collections are transparent to LINKS's query normalization algorithm and we can generate efficient queries even when filtering based on provenance information. This applies to other languages with similarly powerful language-integrated query facilities, notably HASKELL with the DSH library [Ulrich and Grust, 2015].

The above example is aspirational. As yet the depicted language does not quite exist. This dissertation describes three separate languages based on LINKS which explore the design space for language-integrated provenance.

LINKS^{W} (Chapter 3) implements fine-grained where-provenance. Programmers indicate which database tables and columns should carry where-provenance. LINKS^{W} wraps such tables in a view to generate the initial annotations, and then propagates annotated values through the query. Unfortunately, this only works if the query does not do anything interesting with the annotated values except for passing them around. Fortunately, this matches precisely what we expect from the semantics of where-provenance which annotates values that were copied, unchanged, from the database. LINKS^{W} enforces accurate where-provenance annotations through its type system. Where-provenance-annotated values have a special type that distinguishes them from unannotated values and LINKS^{W} guarantees that if a value has provenance type, its annotation is present and accurate. We state this where-provenance correctness property formally and prove that LINKS^{W} propagates annotated values correctly in Theorem 3.5. A consequence of the strongly-typed approach is that if the programmer wants to do anything with an annotated value except for passing it on they have to unwrap it first. The type system is a valuable guide in these cases, but to add where-provenance everywhere in a query can be a bit of effort.

LINKS^{L} (Chapter 4) implements lineage. LINKS^{L} is perhaps the closest to the above hypothetical language. The `lineage` keyword transforms a whole query at once to automatically compute lineage. Unlike in LINKS^{W}, there is no need for programmers to change their query at all except for requesting lineage. In

terms of language design this is a trade-off in precision and control, manual annotation effort, and type-safety. The query translation in Links^{L} is a bit more complicated because it has to deal correctly with uses, in particular in user-defined functions, of lineage-annotated values (unlike Links^{W} which discharges this responsibility to the programmer). Nevertheless, the translation is fairly straightforward thanks to Links's support for queries over nested collections which enables the obvious representation of lineage annotations as multisets.

We discuss the performance of Links^{W} and Links^{L} in Chapter 5, including comparisons to traditional, database-based provenance systems Perm [Glavic, 2010] and Pug [Lee et al., 2018]. The overhead of querying provenance is in line with what we would expect for processing more data and comparable to other systems. Language-integrated provenance generates queries that compute provenance on demand and allow further processing of provenance annotations in the same database query, satisfying Requirements 3 and 4 of Glavic et al. [2013]. The generated queries use the declarative subset of Sql, not user-defined functions, triggers, or other procedural extensions. This gives the database system's query optimizer the opportunity to take both data and provenance into account, even in the presence of complex queries over provenance information.

Links^{T} (Chapter 6) allows users to define their own forms of provenance, including where-provenance and lineage, by writing generic functions that analyze query traces. Links^{W} and Links^{L} are separate extensions of Links, and thus stop short of fulfilling Requirement 1. Even when combined into a single implementation, as Stolarek and Cheney [2018] have done, having to extend the programming language for every new form of provenance a user might be interested in is not ideal. With Links^{T} we explore what generic features we would need to add to the language to be able to implement different forms of provenance as libraries. Built into the language is a self-tracing translation that turns a query into an expression that, when executed, would build a tree-structured trace of the query's execution. However, the intention is not to execute the self-tracing query, but to compose it with a function that analyzes the trace and extracts provenance into a nested relational representation. The language normalizes the composition of a self-tracing query with a trace analysis function to obtain an efficient query that does not actually construct the whole trace.

A trace resembles the shape of the query but also records important data-dependent decisions that were made — or rather will be made — during query

execution, such as whether a filter condition held. The trace datatype is inspired by previous work on program and query slicing [Cheney et al., 2014a; Perera et al., 2012] with two major differences: The traces of lists and records are lists and records of traces; in other words, trace information is accumulated in the leaves of a tree of list and record constructors. This goes hand in hand with not representing variable binding explicitly in the trace. Wherever a bound variable would appear, we replace it with its trace instead. This frees programmers from reimplementing binding in every trace analysis function, which will typically be structured as recursive interpreters of the trace datatype.

Trace analysis functions need to work for queries with any return type. To this end we introduce generic, polytypic programming features in the form of typecase and mapping and folding of records. These are well-known features from other languages such as λ_i^{ML} [Harper and Morrisett, 1995] and Ur/Web [Chlipala, 2010]. However, to be able to compile provenance queries based on the analysis of self-tracing queries to plain Sql, we need to normalize away these metaprogramming features. We extend the Links normalization rules accordingly and prove progress and preservation properties in Section 6.5. Links$^{\mathrm{T}}$ is sufficiently general to implement at least where-provenance and lineage. We show their implementations as trace analysis functions and discuss other forms of provenance in Section 6.3. A prototype implementation of the extended normalization procedure confirms that the generated queries are mostly reasonable despite the increased flexibility.

In summary, this dissertation demonstrates that a programming language can make a fine provenance system all by itself and may even offer increased safety and flexibility over database-integrated provenance systems.

Chapter 2

Background

This chapter includes material from previously published work [Fehrenbach and Cheney, 2016, 2018].

This chapter provides the necessary background to understand the rest of this dissertation. In Section 2.1.1 we describe some forms of provenance that have been identified in the literature. We focus on where-provenance and lineage because they will be used later. Section 2.1.2 reviews database-integrated provenance systems, in particular with respect to the requirements for provenance systems laid out by Glavic, Miller, and Alonso [2013] and discussed in the introduction. Section 2.2 discusses language-integrated query in Links and other systems. Since the rest of this dissertation treats query compilation as a black box we do not go into much detail but point interested readers to the primary literature. Section 2.3 describes the syntax and semantics of a simplified subset of Links. We focus on those aspects of the language that are relevant to understand Links^{W} and Links^{L}. Links^{T} is a bigger deviation from Links, in which polymorphism plays a significant role. It is therefore presented entirely self-contained in Chapter 6 and not based on the description here.

2.1 Provenance in databases

Provenance metadata is information about the origins and history of something. The provenance record of a piece of art includes information about the artist, previous owners, repairs, and restorations. Among other things, a detailed provenance record plays an important role in judging a piece's authenticity. In

informatics, we are interested in the provenance of data and processes. Databases store data and with increasing size and complexity, interest in data provenance grew in the database research community.

Buneman et al. [2000] identify "some basic issues" to do with data provenance. Unlike paper documents, databases can change over time, sometimes rapidly. This has implications for citing work derived from database data, archiving, and reproducibility in general. Provenance metadata can help track sources and modifications. These issues are particularly important where multiple data sources come together like in curated databases, such as those used in molecular biology, and data warehouses which bring together business data stored in different databases for cross-database analysis. However, there is a difference between the provenance of information or knowledge of the sort that is stored in curated scientific databases, data warehouses, and the provenance of specific database query results. Scientific databases store and organize information from different sources. The Gene Ontology [Ashburner et al., 2000; The Gene Ontology Consortium, 2017], for example, stores information about genes, gene products, and cellular functions. This data is ultimately backed by experiments published in peer-reviewed publications. It also stores machine-generated annotations. Thus the provenance of a particular entry may include information like "Inferred from Mutant Phenotype" and link to a study on mutant *drosophila melanogaster* with extra legs in place of antennae. Data warehouses combine information from multiple data sources too, but the sources are usually databases themselves. It may seem easy to point from one database to another (perhaps easier than pointing to experiments on fruit flies), but the source data may change rapidly which brings its own challenges.

The area that has seen the most attention from database researchers is the more narrowly defined *data provenance*. Data provenance is metadata describing the origin of results of a particular database query against a particular database. This narrower focus allows researchers to make precise the meaning of provenance, what exactly it says about the query and data. It turns out that provenance is not one thing, but there are different forms of provenance that are suited for answering different kinds of questions. We review some of the research on different forms of data provenance in Section 2.1.1. We also review some previous implementations of provenance systems in Section 2.1.2.

Data provenance takes the query as a given, and known to be correct. This

is not necessarily the case. Queries might well be too complicated, possibly generated by programs, to be easily understood. In that case, we might want to use provenance to learn about the query and how it transformed the data. Tan [2004] calls this the *provenance of a data product* and Glavic [2010] calls it *transformation provenance*. Provenance is also related to query and program *slicing* [Cheney, 2007; Cheney et al., 2011]. We discuss this particular relationship in more detail in Chapter 6, which implements data provenance on top of tracing and was inspired by work on program and query slicing [Cheney et al., 2014a; Ricciotti et al., 2017]. The sort of provenance supported by the languages described in this dissertation may help in debugging queries but the focus is data. For now, language-integrated provenance stops short of full interactive query debugging. The interested reader may want to consider using tools such as Habitat [Grust and Rittinger, 2013].

For broader surveys of provenance, see Herschel, Diestelkämper, and Ben Lahmar [2017], Simmhan, Plale, and Gannon [2005], and Tan, Ko, and Holmes [2013].

2.1.1 Where? why? how? — questions provenance answers

Given a database D, a query Q, and a piece d of the query result $Q(D)$, the provenance of d should answer the question: "Which parts of D contributed to d?" Buneman, Khanna, and Tan [2001] were the first to point out that this depends crucially on what you mean by "contributed to". They identify two forms of provenance that answer two distinct questions: *where* a piece d was copied from, and *why* it is in $Q(D)$ in the first place. There are other questions one might ask, like *how* or *why-not*? Exact definitions of provenance also depend on details of the data model, the query language, additional properties like minimality and invariance under query rewriting, and implementation considerations.

In the following, we try to give an intuition for some forms of provenance and related concerns with a particular focus on those relevant to Chapters 3, 4, and 6. Cheney, Chiticariu, and Tan [2009a] survey different forms of provenance used in relational databases in greater depth and explain and relate them in a common notational format.

2.1.1.1 Input-provenance

Glavic [2010] categorizes forms of provenance into three groups, the first of which only tracks inputs. In a relational database setting this could be the whole database, or the tables mentioned in a query. This is obviously quite imprecise. The benefit is that one can treat the computation itself as a black box. Input-provenance works well for systems that orchestrate arbitrary computations like generic build systems such as Make [Feldman, 1979] or distributed computing systems such as Spark [Zaharia et al., 2010]. In the database setting input-provenance does not play a big role. Queries are more easily analyzed or traced than arbitrary executables to produce more fine-grained provenance.

2.1.1.2 Where-provenance

Where-provenance is information about *where* information in a query result "came from" (or was copied from) in the input. A common reason for asking for where-provenance is to identify the source of incorrect or surprising data in a query result. For example, if a name in a query result is misspelled we might ask for its where-provenance to find the exact location in the database where we need to fix it.

In the provenance literature, the idea of where-provenance goes back to Buneman et al. [2001] who identify it as distinct from why-provenance. The following is based on a later presentation for the nested relational calculus by Buneman et al. [2008]. In their model, every value is annotated with an abstract *color* from some infinitely large set of distinguishable colors. A table would carry an annotation for the whole table, as would each record or row, and every cell. A query that just returns the whole table would preserve all annotations. A simple filtering query would preserve annotations on unchanged rows and cells, but not the table-level annotation. For example, the table `where-fst-snd` (Table 2.1) is annotated with a and its two rows are annotated with b and c respectively. The individual cells carry annotations d–g.

Where-provenance is arguably the simplest form of provenance but there are some corner cases to consider. The first is that not all values have meaningful where-provenance. Take for example the following query:

```
SELECT fst, snd, fst + snd AS sum, 4 AS const
  FROM where-fst-snd
```

a	fst	snd
b	1^d	2^e
c	2^f	3^g

Table 2.1: Example table `where-fst-snd` with annotations a–g.

The values in columns `fst` and `snd` are copied unchanged and thus should keep their annotations. But how about the values in the `sum` column? They originate from a computation, not the input table. Similarly, the values in column `const` originate not from the input table but a constant in the query itself. Clearly the rows of the result are not copies of any rows in the input either, and neither is the table itself a copy. Buneman et al. [2001] consider the where-provenance of such values to be $\perp$, or undefined. (Provenance systems based on SQL commonly use `NULL` to indicate missing or undefined annotations.) Thus the result with where-provenance annotations is as follows:

$\perp$	fst	snd	sum	const
$\perp$	1^d	2^e	$3^\perp$	$4^\perp$
$\perp$	2^f	3^g	$3^\perp$	$4^\perp$

Missing annotations are not the only challenge. Consider the following two queries which both compute a path from a to b via c.

```
SELECT x.fst AS a, y.snd AS b, x.snd AS c
  FROM where-fst-snd x, where-fst-snd y
 WHERE x.snd == y.fst;

SELECT x.fst AS a, y.snd AS b, y.fst AS c
  FROM where-fst-snd x, where-fst-snd y
 WHERE x.snd == y.fst
```

The queries are identical except for where they take the value for the `c` column from: the left join partner's `snd` column, and the right join partner's `fst` column, respectively. When dealing with unannotated values we consider these queries to be equivalent, because their results are always the same — the condition explicitly states that the left join partner's `snd` column is the same as the right join partner's `fst` column. Looking at the annotations may allow us to distinguish

otherwise equal values. In the definition of where-provenance we have a choice: We can stick with one annotation per value and as a consequence be able to distinguish the first query from the second by whether the c value is 2^e or 2^f, or we can allow multiple annotations and require that all equivalent queries produce the same annotations, which for our example means that the c value should be $2^{\{e,f\}}$. DBNotes [Chiticariu et al., 2005] implements both a simple form of where-provenance called *default* propagation and one that is invariant under query rewriting, called *default-all*. It is not clear that they got query equivalence right, to the point where Gatterbauer, Meliou, and Suciu [2011] say that "default-all is dangerous" because it can return annotations that are irrelevant to the result. Just because some value is equal to some other value does not mean they should share annotations.

For the purposes of this dissertation, the exact details of a given form of where-provenance are not that important, as long as it is implementable by query rewriting. The implementation of where-provenance in Links^W is inspired by Buneman et al. [2008] but annotations are on table cells only. We use multiset semantics, so annotations on equal results are not combined, and neither are annotations from different sources, even when the query compares them for equality, so we are closer to DBNotes's *default* propagation. For details, see Chapter 3 and in particular Theorem 3.5.

2.1.1.3 Why-provenance and lineage

Why-provenance is information that explains *why* a result was produced. In a database query setting, this is usually taken to mean a *justification* or *witness* to the query result, that is, a subset of the input records that includes all of the data needed to generate the result record. Several related forms of why-provenance have been studied [Buneman et al., 2001; Cheney et al., 2009a; Cui et al., 2000; Glavic et al., 2013], but many of them only make sense for set-valued collections.

Lineage is one simple form of why-provenance which is applicable to set and multiset semantics. Intuitively, the lineage of a record r in the result of a query Q is a subset L of the records in the underlying database D that "justifies" or "witnesses" the fact that r is in the result $Q(D)$. That is, running Q on only the records identified as the lineage L of r, should still produce a result containing r, i.e. $r \in Q(L)$. Obviously, this property can be satisfied by many subsets of the input database, including the whole database D, and this is part of the reason

a	b	annotation
1	1	r_1
2	2	r_2

(a) Table r.

a	b	annotation
1	2	s_1
1	5	s_2

(b) Table s.

Figure 2.1: Lineage example database.

why there exist several different definitions of why-provenance (for example, to require minimality). We follow the common approach of defining the lineage to be the set of all input database records accessed in the process of producing r; this is a safe overapproximation of the minimal lineage, and is still usually much smaller than the whole database.

For example, consider the following SQL query.

```
SELECT r.b AS v FROM r, s WHERE r.a = s.a
UNION ALL SELECT s.a AS v FROM s WHERE s.b = 5
```

When run on the annotated database in Figure 2.1 it produces the following lineage-annotated result. We use multiset semantics here, so we have multiple

v	annotations
1	$\{r_1, s_1\}$
1	$\{r_1, s_2\}$
1	$\{s_2\}$

rows with the same value. All rows differ in their annotations. Interestingly, not all rows even have the same number of annotations. The last row is produced by the second argument to **UNION ALL** and has only one row, namely s_2, as its lineage. The number of lineage annotations on a result depends on the structure of the query. Depending on the representation of lineage that a provenance system uses, this can make working with lineage difficult, especially when queries are generated dynamically at runtime. For example, Grust et al. [2004] describe a relational encoding of XML documents and how to translate XPATH expressions into SQL to find nodes. Based on this one can implement an XPATH interpreter using sufficiently powerful language-integrated query facilities [Cheney et al., 2013]. Imagine we wanted to expose this on a website as an educational resource where people can enter XML documents and XPATH queries

and see the result as well as which database rows were touched to produce it (its lineage). In this scenario, the SQL query's shape and therefore the number of lineage annotations depends on the XPATH query which is not known until runtime. PERM, for example, stores lineage in additional columns which makes it hard for the client program to process lineage annotations generically. In LINKS^{L} and LINKS^{T} we use LINKS's support for nested collections in query results to represent lineage annotations.

There is another issue with lineage and the generated XPATH queries, namely emptiness tests and negation. In general, non-monotonic queries, that is queries that use aggregations, emptiness tests, or set difference, do not have well-defined lineage. For example, consider the query that selects everything from table a if table b is empty. Every row in the result would be annotated with a corresponding row in a. So far, so good. However, we would also need to record somehow the fact that b was empty. We could annotate whole tables in addition to individual rows, but this would complicate the annotation model. This is the approach taken in the work on dependency provenance [Cheney et al., 2011] which is similar to lineage but extends to non-monotonic queries. For LINKS^{L}, we chose to only consider monotonic queries (for details on lineage in LINKS^{L} see Chapter 4, in particular Theorem 4.9 and its Corollary 4.10).

Implementations of lineage specifically include WHIPS [Cui and Widom, 2000a; Wiener et al., 1995], TRIO [Agrawal et al., 2006; Benjelloun et al., 2008; Widom, 2005], PERM [Glavic, 2010; Glavic and Alonso, 2009], and the work of Müller et al. [2018]. Lineage can be derived from other forms of why-provenance and it is an instance of semiring provenance as described in the next section.

2.1.1.4 Semiring provenance

Semiring provenance was first proposed by Green, Karvounarakis, and Tannen [2007b]. It captures a variety of provenance definitions in a unified model and thus helped to clarify what exactly people mean when they say provenance, and how their definitions relate to others'.

In the semiring model, values are annotated with elements of a commutative semiring $(K, +, \cdot, 0, 1)$, where K is a set, $+$ is associative, commutative, with identity 0, $\cdot$ is associative, commutative, with identity 1, $\cdot$ distributes over $+$, and multiplication with 0 annihilates. In K-relational algebra, the familiar operators of relational algebra operate on values annotated with elements of K. Operations

a	b	annotation
A	B	r_1
D	F	r_2

(a) Table R.

b	c	annotation
B	C	r_3
D	E	r_4
B	F	r_5

(b) Table S.

a	b	annotation
A	B	$r_1 \cdot r_3 + r_1 \cdot r_5$
D	F	r_2

(c) Result $\pi_{a,b}(R \bowtie S) \cup \sigma_{a=D}(R)$.

Figure 2.2: Semiring provenance example.

like join and Cartesian product combine annotations of their inputs with $\cdot$ and operations like union combine annotations with $+$.

See Figure 2.2 for an example database and an annotated example query result. If we instantiate the semiring as $(\mathbb{B}, \vee, \wedge, \bot, \top)$, we obtain standard set semantics — the annotation reflects whether a tuple is in the result or not. The semiring $(\mathbb{N}, +, \cdot, 0, 1)$ results in multiset semantics — the annotation reflects multiplicity. We can obtain why-provenance in the sense of Buneman et al. [2001] with the following semiring: $(\mathbb{P}(\mathbb{P}(X)), \cup, \Cup, \emptyset, \{\emptyset\})$, where the annotations are sets of sets of some unique tuple identifier X and $A \Cup B = \{a \cup b : a \in A, b \in B\}$ is pairwise union [Green, 2011]. There are other semirings that lead to interesting provenance including one that produces lineage, one that produces annotations like the Trio system, and the most general semiring that produces bags of bags of contributing tuples, or *universal provenance polynomials*, that can be interpreted in a variety of ways to recover the other semirings. However, where-provenance is not quite an instance of semiring provenance because it is defined on the level of cells, not rows [Cheney et al., 2009a]. Conversely, there are semirings that are not exactly forms of data provenance, but where the annotations reflect security levels, possible worlds, or uncertainty.

Semiring provenance has been extended to nested relational calculus and Xml [Foster et al., 2008], aggregations [Amsterdamer et al., 2011b], difference [Amsterdamer et al., 2011a], and negation [Köhler et al., 2013]. For even more on semiring provenance, see recent surveys of the literature [Green and Tannen, 2017; Karvounarakis and Green, 2012].

2.1.2 Implementations

In this section, we briefly describe a few of the growing number of provenance systems. In the introduction, we pointed out some design criteria for provenance systems, including the requirements laid out by Glavic, Miller, and Alonso [2013]. We will refer back to these in the following and point out for each system where it stands with respect to supporting different kinds of provenance, the complexity of the supported query language, the queries supported on provenance metadata itself, and support for large databases. We also discuss implementation strategies, in particular whether a given system works with standard databases. The purpose of this overview is not so much to rank existing systems, but rather to show where language-integrated provenance fits in.

2.1.2.1 Whips

A data warehouse is a big database that stores data from different sources — often themselves databases — in a single place for easier analysis. There are two major components in a data warehouse: query execution and data integration. The former is responsible for executing queries on the stored data. The latter is responsible for fetching data from sources, transforming it, and pushing it to the warehouse for storage.

Whips is a prototype data integration system that "collects data from heterogeneous sources, transforms and summarizes it according to warehouse specifications, and integrates it into the warehouse" [Wiener et al., 1995]. Among other things, it aims to solve a problem that earlier data warehouse systems had: they had to be shut down periodically to integrate new data and recompute materialized views. To this end, Whips tracks lineage: "For a given data item in a materialized warehouse view, we want to identify the set of source data items that produced the view item" [Cui et al., 2000].

Whips is a distributed system implemented in C/C++. Users write queries to define views in Sql, and lineage is tracked internally, automatically by the system. Even though query debugging is mentioned as one use of lineage in some of the work on Whips, there is no support for querying lineage directly in the language. Thus, while Whips implements provenance, it is not really a system to be used for provenance queries by end-users.

A particular concern for Whips is storage of annotations, or even the whole

source data, in the warehouse itself to be able to recompute views without accessing the source databases [Cui and Widom, 2000b]. This is, by design, contrary to Requirement 4 which calls for provenance to be computed on demand only.

2.1.2.2 DBNotes

DBNotes is "a Post-It note system for relational databases where every piece of data may be associated with zero or more notes (or annotations) [which] are transparently propagated along as data is being transformed" [Chiticariu et al., 2005]. It supports three annotation propagation schemes called *default*, *default all*, and *custom*. The *default* scheme is what we call where-provenance. The *default all* scheme is intended as where-provenance that is invariant under query rewriting. It should propagate all annotations that all equivalent formulations of a query would propagate. Unfortunately, this has been shown to sometimes propagate too many annotations [Gatterbauer et al., 2011]. The *custom* scheme allows queries to specify which input cells propagate their annotations to which output cells. A user can encode something like lineage using the *custom* annotation propagation scheme but has to do so carefully for each query as there are no facilities for abstracting over and reusing custom propagation schemes.

The DBNotes implementation consists of two parts: the translator and the postprocessor [Bhagwat et al., 2005]. The translator takes queries in pSql and translates them to one or more Sql queries. These are handed to a database system (Oracle 9i) for evaluation. The postprocessor collects annotations from the flat relational encoding into nested collections for display.

DBNotes supports select-project-join queries on sets with limited support for aggregations. Annotations are stored alongside data in the database, in one annotation column per data column for every relation. This storage scheme makes multiset queries impossible, because a result of two equal tuples with one annotation each, is indistinguishable from one tuple with two annotations.

2.1.2.3 Trio

Trio is "a new database system that manages not only data, but also the accuracy and lineage of the data" [Widom, 2005]. It is not primarily a provenance system, but rather a database for uncertain values that tracks confidence and sources.

It implements lineage as the only form of provenance. Confusingly, the work on Trio uses the word "lineage" more generally for what we call "provenance" and claims that what they are tracking is closer to where-provenance than why-provenance [Benjelloun et al., 2008].

One of the stated goals of Trio is to "deploy a working system that is sufficiently easy to adopt and efficient enough that it actually gets used" [Widom, 2005]. Consequently, its authors chose to implement Trio on top of a standard database system [Benjelloun et al., 2008]. They define their own query language called TriQL, which is based on Sql but extended with features for dealing with uncertainty and lineage. There is a hint of language-integrated provenance in Trio: interaction with Trio happens via an extended version of Python's database interface, which rewrites TriQL queries to one or more Sql queries to pass to PostgreSQL, where the data is stored, for evaluation. However, Trio uses user-defined functions to implement database operations that propagate annotations. User-defined functions are hard for database systems to optimize. Also, lineage is stored in separate tables in the database, violating Requirement 4 of Glavic et al. [2013].

2.1.2.4 Perm

Perm [Glavic, 2010; Glavic and Alonso, 2009; Glavic et al., 2013] is a provenance system implemented on top of PostgreSQL. Given its authors, it is no great surprise that Perm fulfills all of the aforementioned requirements for provenance systems.

Perm supports multiple forms of provenance, including a variation of where-provenance and its own Perm *influence contribution semantics* which is based on lineage. "Perm also supports propagating user-defined annotations" [Glavic et al., 2013]. As in Links^{W} however, this customization is limited to the content of initial annotations; it does not affect their propagation behavior.

Unlike Trio and DBNotes, Perm uses a purely relational representation of provenance annotations. This encoding allows Perm to use standard relational operations to operate efficiently not only on data but also provenance information. Partly as a consequence, Perm supports computing provenance for a large subset of Sql, and it supports all of Sql for operating on provenance metadata. Perm uses additional columns to represent the lineage annotations of each row in a result. The number of columns and their names depend on the structure of

the query which makes them slightly difficult to predict and use correctly.

Perm is a fork of PostgreSQL 8.3. There are minor changes to the parser and other frontend components. The major new component is rewriting provenance queries, just before query planning. Since Perm uses a flat relational representation of provenance data, no postprocessing is necessary. We compare the performance of Links^{W} and Links^{L} to Perm in Chapter 5.

2.1.2.5 GProM

GProM is the prototype "for a generic database provenance middleware" [Arab et al., 2014]. In many ways it is the successor of Perm. It uses a query language that is similar to Perm's and supports Perm's form of provenance. In addition, it supports retroactively computing provenance for past transactions and updates if the database system supports time travel.

GProM is a database middleware, sitting between a client and a database system. Because it rewrites queries before they ever reach the database, it can theoretically support any backend database system. Initially it only supported Oracle, but its extension Pug (see below) supports PostgreSQL as well.

GProM has a special-purpose optimizer for generated provenance queries [Niu et al., 2017]. As Müller et al. [2018] observe, queries generated through provenance rewrites are somewhat untypical for database systems and thus not always executed optimally. Perhaps GProM's and other targeted optimizations can be applied to help database systems with planning provenance queries.

2.1.2.6 Pug

Pug [Lee et al., 2018] is an extension of GProM with a graph-based provenance model on Datalog queries. Their graph-based model is equivalent to provenance games [Köhler et al., 2013], which in turn generalize semiring provenance. From this model it is possible to efficiently extract more compact forms of provenance, including why-provenance and why-not–provenance.

The query language is non-recursive Datalog with negation. Aggregations are not supported. Datalog queries are rewritten to propagate successful and failed rule derivations bottom-up. The resulting instrumented Datalog programs are compiled to Sql and executed on a database system (Oracle or PostgreSQL).

Lee et al. [2018] used Pug's support for lineage to compare it to Links^{L} using some of the benchmark queries from our comparison with Perm (see Chapter 5). For small database instances Links^{L} performs better; for larger instances Pug performs better. This is partly due to the somewhat inefficient in-memory representation of query result data in Links, as their comparison of the raw Sql query runtimes shows.

2.1.2.7 ProQL

ProQL [Karvounarakis et al., 2010] is a language and prototype implementation for data provenance based on a graph representation of semiring provenance [Green et al., 2007a,b]. Provenance data is encoded in relations, which are stored in a relational database. "[The] ProQL prototype, including parsing, unfolding and translation to Sql queries was implemented as a Java layer running atop" Db2 [Karvounarakis et al., 2010]. Unlike previous systems, there is some focus on investigating the use of indices to speed up provenance queries.

2.1.2.8 (Mis-)interpreting Sql

Müller, Dietrich, and Grust [2018] propose a "compositional source-level Sql rewrite" that transforms an input query "into its own interpreter that wields data dependencies instead of regular values". This work covers a richer subset of Sql than any other provenance system described here, "including recursion, windowed aggregates, and user-defined functions" [Müller et al., 2018].

Generating provenance for a query q works in two phases. In the first phase, they run an instrumented query $q^{\mathbb{1}}$ that produces the same result as q but also records value-based decisions, like whether a predicate held for a row, as a side effect of evaluation. In the second phase, a query q^{2} replays decisions made and recorded by $q^{\mathbb{1}}$ and returns dependency sets. These dependency sets encode where-provenance and a form of cell-level why-provenance. Both the tracing query $q^{\mathbb{1}}$ and the interpretation query q^{2} are Sql queries, so this approach requires no changes to the database system. However, the queries make heavy use of advanced Sql features. In theory they could work on any database; in practice the authors used only PostgreSQL. Performance is better than Perm on complex queries with large results and worse than Perm on simple queries with small results. Since provenance is only calculated after the fact

from a trace of the original query's execution, provenance metadata cannot be used directly inside the query.

2.1.2.9 ProvSQL

ProvSQL is "an open-source module for the PostgreSQL database management system that adds support for computation of provenance and probabilities of query results" [Senellart et al., 2018]. It supports where-provenance, semiring provenance, and *m*-semiring provenance (for negation and why-not–provenance). Unlike some of the other systems that claim semiring provenance support, the user can actually define their own semiring and ProvSQL will use it to compute annotations. Thus a user does not have to extract their custom provenance from a general provenance polynomial.

Unlike Perm, ProvSQL is not a fork of PostgreSQL, but rather a *module* which builds on explicit extension mechanisms. As such it should be relatively easy to add to an existing PostgreSQL installation. However, it is more tightly integrated into the database than middleware systems like GProM and systems based on rewriting to plain Sql queries.

Because of its implementation as a module, ProvSQL is restricted in its interface. In particular, there are no language extensions for querying provenance. All provenance-related operations are exposed as user-defined functions.

2.2 Language-integrated query

Writing programs that interact with databases can be tricky, because of mismatches between the models of computation and data structures used in databases and those used in conventional programming languages. The default solution to accessing a database from a program (employed by Jdbc and other typical database interface libraries) is for the programmer to write queries or other database commands as uninterpreted strings in the host language, and these are sent to the database to be executed. This means that the types and names of fields in the query cannot be checked at compile time and any errors will only be discovered as a result of a runtime crash or exception. More insidiously, failure to adequately sanitize user-provided parameters in queries opens the door to Sql injection attacks [Shar and Tan, 2013].

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

Figure 2.3: Example tour agencies database instance.

Language-integrated query is a technique for embedding queries into the host programming language so that their types can be checked statically and parameters are automatically sanitized. Broadly, there are two common approaches to language-integrated query. The first approach, which we call Sql *embedding*, adds specialized constructs resembling Sql queries to the host language, so that they can be typechecked and handled correctly by the program. This is the approach taken in C# [Meijer et al., 2006], Sml# [Ohori and Ueno, 2011], and Ur/Web [Chlipala, 2015]. The second approach, which we call *comprehension*, uses monadic comprehensions or related constructs of the host language, and generates queries from such expressions. The comprehension approach builds on foundations for querying databases using comprehensions developed by Buneman et al. [1995], and has been adopted in languages such as F# [Syme, 2006] and Links [Cooper et al., 2007] as well as libraries such as Database-Supported Haskell (Dsh) [Giorgidze et al., 2011; Ulrich and Grust, 2015] and QueΛ [Suzuki et al., 2016].

The advantage of the comprehension approach over Sql embedding is that it provides a higher level of abstraction for programmers to write queries, without sacrificing performance. This advantage is critical to our work, so we will explain it in some detail. For example, the query shown in Figure 2.4 illustrates

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

Figure 2.4: Boat tours query.

the use of Links's comprehension syntax. It asks for the names and phone numbers of all agencies having an external tour of type *"boat"*. The keyword **for** performs a comprehension over a table (or other collection), and the **where** keyword imposes a Boolean condition filtering the results. The result of each iteration of the comprehension is a singleton collection containing the record `(name = e.name,phone = a.phone)`.

Monadic comprehensions do not always correspond exactly to Sql queries, but for queries that map flat database tables to flat results, it is possible to normalize these comprehension expressions to a form that is easily translatable to Sql [Wong, 1996]. For example, the boat tours query in Figure 2.4 does not directly correspond to a Sql query due to the alternation of **for** and **where** operations; nevertheless, query normalization generates a single equivalent Sql query in which the **where** conditions are both pushed into the Sql query's **WHERE** clause:

```
SELECT e.name AS name, a.phone AS phone
  FROM ExternalTours e, Agencies a
 WHERE e.type = 'boat' AND a.name = e.name
```

Normalization frees the programmer to write queries in more natural ways, rather than having to fit the query into a pre-defined template expected by Sql.

However, this freedom blurs the boundary between general purpose programming language and query language and can lead to problems, for example if the programmer writes a query-like expression that contains an operation, such as `print` or regular expression matching, that cannot be performed on the database. In early versions of Links, this could lead to unpredictable performance, because queries would unexpectedly be executed on the server in memory instead of inside the database. The current version uses a type-and-effect system (as described by Cooper [2009] and Lindley and Cheney [2012])

to track which parts of the program must be executed in the host language and which parts may be executed on the database. Using the **query** keyword above forces the typechecker to check that the code inside the braces will successfully execute on the database.

Although comprehension-based language-integrated query may seem (at first glance) to be little more than a notational convenience, it has since been extended to provide even greater flexibility to programmers without sacrificing performance. The original results on normalization (due to Wong [1996]) handle queries over flat input tables, produce flat result tables, and do not allow calling user-defined functions inside queries. Subsequent work has shown how to support higher-order functions [Cooper, 2009; Grust and Ulrich, 2013] and queries that construct nested collections [Cheney et al., 2014c; Ulrich and Grust, 2015]. For example, we can use functions to factor the previous query into reusable components, provided the functions are non-recursive and only perform operations that are allowed in the database.

```
fun matchingAgencies(name) {
  for (a <-- agencies)
  where (a.name == name)
    [(name = e.name, phone = a.phone)] }

query {
  for (e <-- externalTours)
  where (e.type == "boat")
    matchingAgencies(e.name) }
```

Cooper's results show that these queries still normalize to SQL-equivalent queries, and this algorithm is implemented in LINKS. Similarly, we can write queries whose result type is an arbitrary combination of record and collection types, not just a flat collection of records of base types as supported by SQL:

```
query {
  for (a <-- agencies)
    [(name = a.name,
      tours = for (e <-- externalTours)
              where (e.name == a.name)
                [(dest = e.destination, type = e.type)] }
```

This query produces records whose second `tours` component is itself a collection — that is, the query result is of type `[(name:String, [(dest:String, type:Type)])]` which contains a nested occurrence of the collection type con-

structor `[]`. Sql does not directly support queries producing such nested results — it requires flat inputs and query results.[1]

Query shredding is an extension of the Links query normalization algorithm that can translate queries with nested results to Sql [Cheney et al., 2014c]. Given a query whose return type contains n occurrences of the collection type constructor, query shredding generates n Sql queries that can be evaluated on the database. Some post-processing is necessary to construct the nested result from the resulting tables. Still, this is typically much more efficient than loading the database data into memory and evaluating the query there.

Both capabilities — higher-order functions and nested query results — are essential building blocks for our approach to provenance. They enable a fairly naive, compositional translation to produce efficient enough queries. Their concrete implementation does not matter much. The rest of this dissertation will treat query compilation as a black box. Other approaches to nested queries include *loop lifting* [Grust et al., 2010] which powered Ferry and an experimental branch of Links [Ulrich, 2011]; *flattening* which is used in Database-supported Haskell (Dsh) [Ulrich and Grust, 2015]; and QueΛ [Suzuki et al., 2016] which implements query shredding plus some extensions in OCaml. Links implements query shredding and being extensions of Links, that is what the languages in Chapters 3, 4, and 6 use, but any of the other approaches to nested queries would work. Being based on Links, this work inherits some of its limitations like no support for grouping, aggregations, and mixed set and multiset semantics.

2.3 Links syntax & semantics

Links is a research language for "web programming without tiers" [Cooper et al., 2007]. The traditional architecture for web applications is in three tiers: server, client, and database. Parts of what is conceptually a single web application are written in different languages and run on different machines. The server is written in Perl, Php, Python, JavaScript, Java, Ruby, etc. and sits in the middle to generate pages (Html, Css, JavaScript) to send to the browser and queries (Sql) to send to the database. Ensuring consistency across language barriers is difficult. One solution to this problem is to write the whole web application in a

[1] Sql:1999 standardizes the `ARRAY` type, but arrays are not relations and come with their own restrictions.

single language: Links.

Links is a statically typed functional programming language. Functions can be annotated to indicate that they should run on the client or the server. Server functions are run by the Links interpreter. Links translates client functions to JavaScript to be run in the browser. Client functions can call server functions and vice versa seamlessly. Concurrent Links programs can communicate via message passing. To access the database, Links uses comprehensions over database tables. These are just like list comprehensions, but are translated to Sql queries and executed on the database. The Links type system ensures that the server, client, and database parts all fit together.

Figure 2.5 presents a simplified subset of Links syntax, sufficient for explaining the provenance translations in Chapters 3 and 4. Types include base types O (such as integers, booleans and strings), table types $\texttt{table}\,(l_i\texttt{:}\ A_i)^{\,n}_{i=1}$, function types $A\ \texttt{->}\ B$, record types $(l_i\texttt{:}\ A_i)^{\,n}_{i=1}$, and collection types $\texttt{[}A\texttt{]}$. In Links, collection types are treated as multisets inside database queries (reflecting Sql's default multiset semantics), but represented as lists during ordinary execution. We use concat lists instead of the more typical cons lists in the formalization because they lead to nicer rules for comprehensions.

Expressions include standard constructs such as constants, variables, record construction and field projection, conditionals, and n-ary recursive functions and application. We use pair types (A,B) and pair syntax (M,N) and projections $M.1$, $M.2$ etc., which are easily definable using records. Constants c can be functions such as integer addition, equality tests, etc.; their types are collected in a signature Σ. The signature Σ is also a simple model of a database: it maps tables to their contents. In Links we write $\texttt{var}\ x = M\texttt{;}N$ for binding a variable x to the value of M in expression N. The semantics of the Links constructs discussed so far is call-by-value. The expression $\texttt{query}\ \{M\}$ introduces a query block, whose content is not evaluated in the usual call-by-value fashion but instead first *normalized* to a form equivalent to an Sql query, and then submitted to the database server. The resulting table (or tables, in the case of a nested query result) are then translated into a Links value. Queries can be constructed using the expressions for the empty collection $\texttt{[]}$, singleton collection $\texttt{[}M\texttt{]}$, and concatenation of collections $M \texttt{++} N$. In addition, the comprehension expressions $\texttt{for}\,(\texttt{x}\ \texttt{<--}\ M)\,N$ and $\texttt{for}\,(\texttt{x}\ \texttt{<-}\ M)\,L$ allow us to form queries involving iteration over tables and collections. The difference between the two

Base types	O	::=	`Int` | `Bool` | `String`
Rows	R	::=	$\cdot \mid R, l : A$
Table types	T	::=	**table**(R)
Types	A, B	::=	$O \mid T \mid A \texttt{->} B \mid \texttt{(}R\texttt{)} \mid \texttt{[}A\texttt{]}$
Contexts	Γ	::=	$\cdot \mid \Gamma, x : A$
Expressions	L, M, N	::=	$c \mid x \mid (l_i = M_i)^n_{i=1} \mid N.l$
		|	**fun** $f(x_i\|^n_{i=0})\ N \mid N(M_i\|^n_{i=0})$
		|	**var** $x = M; N$ | **if** $(L)\ \{M\}$ **else** $\{N\}$
		|	**query** $\{N\}$ | **table** *name* **with** $(l_i : O_i)^n_{i=1}$
		|	`[]` | `[`N`]` | N `++` M | **empty**(M)
		|	**for** $(x$ `<-` $L)\ M$ | **where**$(M)\ N$
		|	**for** $(x$ `<--` $L)\ M$ | **insert** L **values** M
		|	**update** $(x$ `<--` $L)$ **where** M **set** N
		|	**delete** $(x$ `<--` $L)$ **where** M
Values	V, W	::=	$c \mid (l_i = V_i)^n_{i=1} \mid$ **fun** $f(x_i\|^n_{i=0})\ N \mid [] \mid [V] \mid V +\!\!+ W$

Figure 2.5: Syntax of a subset of Links.

expressions is that **for** $(x$ `<--` $M)$ expects M to be a table reference, whereas **for** $(x$ `<-` $M)$ expects M to be a collection. The expression **where** $(M)\ N$ is equivalent to **if** $(M)\ \{N\}$ **else** $\{$`[]`$\}$, and is intended for use in filtering query results. The expression **empty** (M) tests whether the collection produced by M is empty. These comprehension syntax constructs can also be used outside a query block, but they are not guaranteed to be translated to queries in that case. The **insert**, **delete** and **update** expressions perform updates on database tables; they are implemented by direct translation to the analogous Sql update operations.

Figure 2.6 presents the evaluation judgment $\Sigma, M \to \Sigma', M'$ for Links expressions. We employ evaluation contexts $\mathcal{E}$ (following Felleisen and Hieb [1992]) and define the semantics using several axioms that handle redexes and a single inference rule that shows how to evaluate an expression in which a redex occurs inside an evaluation context. The rule for **update** uses syntactic sugar for record update called **with** for brevity. Most of the rules in Figure 2.6 are pure in the sense that they have no side-effect on the state of the database. Only the rules for **insert**, **delete** and **update** may change the database state. The

$$\Sigma, (\mathbf{fun}\ f(x_i|_{i=0}^n)\ M)(V_i|_{i=0}^n) \longrightarrow \Sigma, M[f := \mathbf{fun}\ f(x_i)\ M, x_i := V_i]$$

$$\Sigma, \mathbf{var}\ x = V; M \longrightarrow \Sigma, M[x := V]$$

$$\Sigma, (l_i = V_i)_{i=1}^n.l_k \longrightarrow \Sigma, V_k$$

$$\Sigma, \mathbf{if}\ (\mathbf{true})\ M\ \mathbf{else}\ N \longrightarrow \Sigma, M$$

$$\Sigma, \mathbf{if}\ (\mathbf{false})\ M\ \mathbf{else}\ N \longrightarrow \Sigma, N$$

$$\Sigma, \mathbf{query}\ M \longrightarrow \Sigma, M$$

$$\Sigma, \mathbf{empty}([\,]) \longrightarrow \Sigma, \mathbf{true}$$

$$\Sigma, \mathbf{empty}(V) \longrightarrow \Sigma, \mathbf{false} \quad \text{iff } V \neq [\,]$$

$$\Sigma, \mathbf{for}\ (x <\!\!- [\,])\ M \longrightarrow \Sigma, [\,]$$

$$\Sigma, \mathbf{for}\ (x <\!\!- [V])\ M \longrightarrow \Sigma, M[x := V]$$

$$\Sigma, \mathbf{for}\ (x <\!\!- V +\!\!+ W)\ M \longrightarrow \Sigma, (\mathbf{for}\ (x <\!\!- V)\ M) +\!\!+ (\mathbf{for}\ (x <\!\!- W)\ M)$$

$$\Sigma, \mathbf{for}\ (x <\!\!-\!\!- \mathbf{table}\ t)\ M \longrightarrow \Sigma, \mathbf{for}\ (x <\!\!- \Sigma(t))\ M$$

$$\Sigma, \mathbf{insert}\ (\mathbf{table}\ t)\ \mathbf{values}\ V \longrightarrow \Sigma[t \mapsto \Sigma(t) +\!\!+ V], ()$$

$$\frac{\Sigma' = \Sigma[t \mapsto [X \in \Sigma(t) | \Sigma, M[x := X] \longrightarrow^* \Sigma, \mathbf{false}]]}{\Sigma, \mathbf{delete}\ (x <\!\!-\!\!- \mathbf{table}\ t)\ \mathbf{where}\ M \longrightarrow \Sigma', ()}$$

$$\frac{\Sigma' = \Sigma[t \mapsto [u(X) | X \in \Sigma(t)]] \quad u(X) = \begin{cases} (X\ \mathbf{with}\ l_i = V_i) & \text{if } M[x := X] \longrightarrow^* \mathbf{true} \\ & \text{and } N_i[x := X] \longrightarrow^* V_i \\ X & \text{otherwise} \end{cases}}{\Sigma, \mathbf{update}\ (x <\!\!-\!\!- \mathbf{table}\ t)\ \mathbf{where}\ M\ \mathbf{set}\ (l_i = N_i)_{i=1}^n \longrightarrow \Sigma', ()}$$

$$\frac{\Sigma, M \longrightarrow \Sigma', M'}{\Sigma, \mathcal{E}[M] \longrightarrow \Sigma', \mathcal{E}[M']}$$

$$\begin{array}{rcl} \mathcal{E} & ::= & [] \mid \mathcal{E}(M_1, \ldots, M_n) \mid V(V_1, \ldots, V_{i-1}, \mathcal{E}, M_{i+1}, \ldots, M_n) \\ & \mid & (l_1 = V_1, \ldots, l_{i-1} = V_{i-1}, l_i = \mathcal{E}, l_{i+1} = M_{i+1}, \ldots, l_n = M_n) \mid \mathcal{E}.l \\ & \mid & \mathbf{if}\ (\mathcal{E})\ M\ \mathbf{else}\ N \mid \mathbf{empty}(\mathcal{E}) \mid [\mathcal{E}] \mid \mathcal{E} +\!\!+ M \mid V +\!\!+ \mathcal{E} \\ & \mid & \mathbf{for}\ (x <\!\!- \mathcal{E})\ M \mid \mathbf{for}\ (x <\!\!-\!\!- \mathcal{E})\ M \\ & \mid & \mathbf{insert}\ (\mathcal{E})\ M \mid \mathbf{insert}\ (\mathbf{table}\ t)\ \mathcal{E} \\ & \mid & \mathbf{update}\ (x <\!\!-\!\!- \mathcal{E})\ \mathbf{where}\ M\ \mathbf{set}\ (l_i = N_i)_{i=1}^n \\ & \mid & \mathbf{delete}\ (x <\!\!-\!\!- \mathcal{E})\ \mathbf{where}\ M \end{array}$$

Figure 2.6: Semantics of Links.

rules here present the semantics of Links at a high level, and do not model the compilation to Sql queries; instead **query** $\{M\}$ just evaluates to M.

The type system (again a simplification of the full system) is illustrated in Figure 2.7. Many rules are standard; we assume a typing signature Σ mapping constants and primitive operations to their types. The rule for **query** $\{M\}$ refers to an auxiliary judgment $A :: \mathsf{QType}$ that essentially checks that A is a valid query result type, meaning that it is constructed using base types and collection or record type constructors only:

$$\frac{}{O :: \mathsf{QType}} \qquad \frac{[A_i :: \mathsf{QType}]_{i=1}^{n}}{(l_i : A_i)_{i=1}^{n} :: \mathsf{QType}} \qquad \frac{A :: \mathsf{QType}}{[A] :: \mathsf{QType}}$$

Similarly, the $R :: \mathsf{BaseRow}$ judgment is used in the Table-rule to ensure that the types used in a row are all base types:

$$\frac{}{\cdot :: \mathsf{BaseRow}} \qquad \frac{R :: \mathsf{BaseRow}}{R, l : O :: \mathsf{BaseRow}}$$

Links is a research language and includes a number of features that are not needed to understand the rest of this dissertation such as variants, recursive datatypes, Xml literals, client/server annotations, formlets [Cooper et al., 2008], algebraic effects and handlers [Hillerström and Lindley, 2016], session types [Fowler et al., 2019; Lindley and Morris, 2017], and incremental relational lenses [Horn et al., 2018]. Since most of these features do not interact with the language-integrated query parts of Links, we ignore them. Even in the query parts, the core language of Links described here is a significant simplification of the full language. To determine whether the code inside a **query** block is translatable to Sql, Links uses a type-and-effect system [Lindley and Cheney, 2012]. We use a simplified version of Links's type system that leaves out effects and polymorphism. We assume that constants are pure and have a database equivalent (e.g. no `print`) and that functions are non-recursive. We also assume that database updates do not appear in queries. This simplified model is sufficient to explain the contributions of Chapters 3 and 4. The prototype implementations of Links$^{\mathrm{W}}$ and Links$^{\mathrm{L}}$ are extensions of Links and support the full language (with some restrictions mentioned later). Polymorphism and recursive functions play an important role in Chapter 6 where we use a different core language.

$$\textsc{Const}\ \frac{\Sigma(c) = A}{\Gamma \vdash c : A} \qquad \textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Record}\ \frac{\Gamma \vdash M_i : A_i \quad (i \in \{1,\ldots,n\})}{\Gamma \vdash (l_i = M_i)_{i=1}^{n} : (l_i : A_i)_{i=1}^{n}} \qquad \textsc{Projection}\ \frac{\Gamma \vdash M : (l_i : A_i)_{i=1}^{n}}{\Gamma \vdash M.l_k : A_k}$$

$$\textsc{Fun}\ \frac{\Gamma, [x_i : A_i]_{i=1}^{n} \vdash M : B}{\Gamma \vdash \mathbf{fun}\ (x_i|_{i=1}^{n})\ \{M\} : (A_i|_{i=1}^{n})\ \text{->}\ B}$$

$$\textsc{App}\ \frac{\Gamma \vdash M : (A_i|_{i=1}^{n})\ \text{->}\ B \qquad \Gamma \vdash N_i : A_i \quad (i \in \{1,\ldots,n\})}{\Gamma \vdash M(N_i|_{i=1}^{n}) : B} \qquad \textsc{Var}\ \frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \mathbf{var}\ x\ = M; N : B}$$

$$\textsc{Query}\ \frac{\Gamma \vdash M : [A] \qquad A :: \mathsf{QType}}{\Gamma \vdash \mathbf{query}\ \{M\} : [A]} \qquad \textsc{Empty}\ \frac{\Gamma \vdash M : [A]}{\Gamma \vdash \mathbf{empty}(M) : \mathtt{Bool}} \qquad \textsc{Table}\ \frac{R :: \mathsf{BaseRow}}{\Gamma \vdash \mathbf{table}\ n\ \mathbf{with}\ (R) : \mathbf{table}(R)}$$

$$\textsc{Empty-List}\ \frac{}{\Gamma \vdash [\,] : [A]} \qquad \textsc{List}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : [A]} \qquad \textsc{Concat}\ \frac{\Gamma \vdash M : [A] \qquad \Gamma \vdash N : [A]}{\Gamma \vdash M\ \text{++}\ N : [A]}$$

$$\textsc{For-List}\ \frac{\Gamma \vdash L : [A] \qquad \Gamma, x : A \vdash M : [B]}{\Gamma \vdash \mathbf{for}\ (x\ \text{<-}\ L)\ M : [B]} \qquad \textsc{Where}\ \frac{\Gamma \vdash M : \mathtt{Bool} \qquad \Gamma \vdash N : [B]}{\Gamma \vdash \mathbf{where}\ (M)\ N : [B]}$$

$$\textsc{For-Table}\ \frac{\Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma, x : (R) \vdash M : [B]}{\Gamma \vdash \mathbf{for}\ (x\ \text{<--}\ L)\ M : [B]} \qquad \textsc{Insert}\ \frac{\Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma \vdash M : [\,(R)\,]}{\Gamma \vdash \mathbf{insert}\ L\ \mathbf{values}\ M : ()}$$

$$\textsc{Update}\ \frac{\Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma, x : (R) \vdash M : \mathtt{Bool} \qquad \Gamma, x : (R) \vdash N : (R)}{\Gamma \vdash \mathbf{update}\ (x\ \text{<--}\ L)\ \mathbf{where}\ M\ \mathbf{set}\ N : ()}$$

$$\textsc{Delete}\ \frac{\Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma, x : (R) \vdash M : \mathtt{Bool}}{\Gamma \vdash \mathbf{delete}\ (x\ \text{<--}\ L)\ \mathbf{where}\ M : ()}$$

Figure 2.7: Typing rules for Links.

Chapter 3

Links^W — where-provenance in Links

This chapter includes material from previously published work [Fehrenbach and Cheney, 2015, 2016, 2018].

This chapter describes Links^W, a programming language with built-in support for generating and querying well-typed where-provenance. We give a brief overview of the language in Section 3.1. Section 3.2 describes syntax and semantics in detail, including a proof that Links^W propagates where-provenance annotations correctly. In Section 3.3 we discuss one possible implementation strategy: a type-directed translation from Links^W to plain Links. We conclude the chapter by discussing some shortcomings of the design in Section 3.4. We evaluate the performance of a prototype implementation in Chapter 5.

3.1 Overview

In Links^W we indicate which values should carry where-provenance annotations by annotating **`table`** expressions like the one below.

```
var presidentsWhereProvDefault =
  table "presidents"
  with (nth: Int, name: String)
  where nth prov default, name prov default;
```

The last two lines indicate to Links^W that the values in both the `nth` and `name` columns should carry annotations of the form (R, f, i) where R is the source

table (in this case *"presidents"*), f is the column name (*"nth"* or *"name"*), and i is the row number. By default we use POSTGRESQL's `oid` column as the source of row numbers because they can be automatically generated for all tables by the database system.[1] The resulting table could, for example, look like Table 3.1 where annotations are printed as superscripts.

nth	name
⋮	⋮
$44^{(\text{presidents,nth,524})}$	Barack Obama$^{(\text{presidents,name,524})}$
$45^{(\text{presidents,nth, 93})}$	Donald Trump$^{(\text{presidents,name, 93})}$

Table 3.1: What `presidentsWhereProvDefault` could look like.

LINKS^W allows limited customization of initial where-provenance annotations. Instead of writing **`default`**, we can provide a function that is used to compute initial annotations. For example, if we trust the `nth` column to be a key, we might want to identify presidents by that. In that case we could write a table declaration like the one below.

```
var presidentsWhereNth =
  table "presidents" with (nth: Int, name: String)
  where name prov fun (p) { ("presidents", "name", p.nth) };
```

Here we do not put an annotation on values in the column `nth`. More importantly, we use a custom function to annotate values in the column `name`. This function takes a row of the table as input and produces a where-provenance triple. Here we deviate from the default provenance only in using the running number of presidents as the third component. The resulting table could look something like Table 3.2.

Perhaps the most important aspect of where-provenance in LINKS^W is that annotated values are distinguished from plain values by their type. A value of base type `O` that is annotated with where-provenance has type `Prov(O)`. For example, when reading the table declared above into memory, we would get rows where the `name` field has type `Prov(String)` instead of `String`.

```
asList(presidentsWhereNth) : [(nth: Int, name: Prov(String))]
```

[1] Unfortunately, OIDs are 32 bits only and thus not guaranteed to be unique for large tables in POSTGRESQL 10. User-defined annotations can be used to work around this limitation.

nth	name
⋮	⋮
44	Barack Obama$^{(\text{presidents,name,44})}$
45	Donald Trump$^{(\text{presidents,name,45})}$

Table 3.2: What `presidentsWhereNth` could look like.

Annotated values of type `Prov(O)` support two operations: we can use **`data`** to get the underlying data or **`prov`** to get the annotation. These operations are keywords, but can be thought of as functions of type `(Prov(O)) -> O` and `(Prov(O)) -> (String, String, Int)`, respectively. For example, the following query is equivalent to just using the `presidents` table directly, but uses the third component of the custom where-provenance annotation of the `name` field, which is equivalent to `p.nth`.

```
for (p <-- presidentsWhereNth)
  [(name = data p.name, nth = (prov p.name).3)]
```

`Prov` is an abstract type, and while it is isomorphic to a simple pair, there is no constructor for values of `Prov` type. The only way to get a value of `Prov` type is to query an annotated table. This guarantees that where-provenance annotations are accurate. It is impossible to associate provenance metadata with the wrong data by accident or design. For example, the following query would not type-check, because `p.name` has type `Prov(String)`, meaning it is backed by evidence from a database table, but *"Hillary Clinton"* is just a literal `String`.

```
for (p <-- presidentsWhereNth)
  [ if (p.nth <> 45) { p.name } else { "Hillary Clinton" } ]
# This does not have type [Prov(String)]. As much as we might
# want it to be true, the type system does not allow us to lie.
```

In other words, we use the type system to guarantee that every annotated value, that is every value with a `Prov` type, has a valid annotation. In this we differ from previous work: Buneman et al. [2008] for example use the annotation $\perp$ for values originating from the query itself; sometimes annotations are sets and can simply be empty; and SQL-based implementations frequently use `NULL`. This is a trade-off — we favour the guarantee of meaningful annotations over the convenience of a default. Note that it is still possible to write queries like the

one below, which has accurate provenance for *almost* all values.

```
[(name     = "Donald Trump",
  trips    = 999999999, trips_p = ("facts", "alternative", 1))] ++
for (p <-- presidentsWhereNth) for (m <-- metroWhereTrips)
where (p.inauguration == m.date && m.time == 11 && p.nth <> 45)
  [(name     = data p.name,
    trips    = data p.trips, trips_p = prov p.trips)]
```

So far we have discussed where-provenance from a bottom-up perspective: we declare which columns to annotate and then write queries using provenance annotations. One of the most important use cases for where-provenance is debugging databases, that is finding wrong data. In that scenario, we would take an existing query and modify it to return provenance annotations. For example, we might have noticed a misspelled name or a number that seems too good to be true. We would take the query expression and add a column, next to the one with the wrong data, that returns the provenance. By using the **prov** keyword we force inference of a `Prov` type and follow the type errors until we either reach a table column to annotate, or an expression that cannot possibly have where-provenance, like a constant.

3.2 Syntax & semantics

The syntax of plain LINKS as shown in Figure 2.5 on page 31 is extended as follows:

Values	V	::=	$\cdots \mid V^c$
Types	O	::=	$\cdots \mid \mathbf{Prov}(O)$
Terms	L, M, N	::=	$\cdots \mid \mathbf{data}\ M \mid \mathbf{prov}\ M \mid \mathbf{table}\ n\ \mathbf{with}\ (R)\ \mathbf{where}\ S$
Provenance specifications	S	::=	$\cdot \mid S, l\ \mathbf{prov}\ M$

Values V can be annotated with a color c to produce annotated values V^c. Since the annotation propagation behavior is independent of the values of annotations it is often defined using abstract annotations from an infinite set of distinguishable colors [Buneman et al., 2008; Chiticariu et al., 2005]. In LINKS^W annotations have a meaning and can be inspected. We fix colors to be triples (R, f, i) and interpret R as the source table name, f as the field name, and i as the row identifier. Annotated values have type $\mathbf{Prov}(O)$, where O is a type argument of base type. We treat $\mathbf{Prov}(O)$ itself as a base type, so that it can be used as part of a table type. (This is needed for initializing provenance as

explained below.) For example, $42^{(QA,a,23)}$ has type $\mathbf{Prov}(O)$ and represents the query result 42 which was copied from row 23, column a, of table QA. Crucially, plain values cannot be annotated freely by source programs; only the Links^{W} runtime can construct them.

We add two additional operations, **prov** and **data**, to extract the provenance annotation and the value itself, respectively, from an annotated value. We extend the semantics from Figure 2.6 on page 32 with the following rules.

$$\begin{array}{rcl} \Sigma, \mathbf{prov}\, V^c & \longrightarrow & \Sigma, c \\ \Sigma, \mathbf{data}\, V^c & \longrightarrow & \Sigma, V \end{array}$$

$$\mathcal{E} \quad ::= \quad \cdots \mid \mathbf{prov}\, \mathcal{E} \mid \mathbf{data}\, \mathcal{E}$$

The Links^{W} runtime constructs initial annotations for database values. We allow programmers to indicate which columns in a database table should carry annotations and give some control over what the annotations themselves are. To this end, we extend the syntax of table expressions to allow a list of *provenance initialization specifications* l **prov** M. The expression M in a provenance initialization specification is a function from the type of the unannotated table row to a provenance triple `(String,String,Int)`. A column need not be annotated with provenance at all. In place of M, we can write **default**, as syntactic sugar for a function of the form **fun**`(r){(`T`,`C`,r.oid)}` where T and C are replaced by the table and column name, respectively. For example, if we added default where-provenance to the phone field of the Agencies table, we would execute the following function on every row, to obtain the phone number's provenance: **fun**`(a){("Agencies","phone",a.oid)}`. This way we have three different kinds of columns: plain columns without annotations; columns with *default* where-provenance where the annotation will be the table name, column name, and the row's `oid`; and columns with annotations that are computed by some user-defined function that takes the table row as input.

The typing rules for the new constructs of Links^{W} are shown in Figure 3.1. The rules for **prov** and **data** are as one would expect from the semantics. The table rule employs an auxiliary judgment $\Gamma \vdash S : \mathsf{ProvSpec}(R)$, meaning that in context Γ, the provenance specification S is valid with respect to record type R. It also uses an auxiliary operation $R \triangleright S$ that defines the type of the provenance view of a table whose fields are described by R and whose provenance specification

$$\frac{\Gamma \vdash V : O \qquad \Gamma \vdash c : \texttt{(String, String, Int)}}{\Gamma \vdash V^{c} : \mathbf{Prov}(O)}$$

$$\frac{\Gamma \vdash M : \mathbf{Prov}(O)}{\Gamma \vdash \mathbf{prov}\, M : \texttt{(String, String, Int)}} \qquad \frac{\Gamma \vdash M : \mathbf{Prov}(O)}{\Gamma \vdash \mathbf{data}\, M : O}$$

$$\frac{R :: \mathsf{BaseRow} \qquad \Gamma \vdash S : \mathsf{ProvSpec}(R)}{\Gamma \vdash \mathbf{table}\, n\, \mathbf{with}\ (R)\ \mathbf{where}\, S : \mathbf{table}\,(R \triangleright S)} \qquad \frac{\Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma \vdash M : \texttt{[(}\lfloor R \rfloor\texttt{)]}}{\Gamma \vdash \mathbf{insert}\, L\, \mathbf{values}\, M : \texttt{()}}$$

$$\frac{\Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma, x : \texttt{(}\lfloor R \rfloor\texttt{)} \vdash M : \texttt{Bool} \qquad \Gamma, x : \texttt{(}\lfloor R \rfloor\texttt{)} \vdash N : \texttt{(}\lfloor R \rfloor\texttt{)}}{\Gamma \vdash \mathbf{update}\ (x \texttt{<--}\, L)\ \mathbf{where}\, M\ \mathbf{set}\, N : \texttt{()}}$$

$$\frac{\Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma, x : \texttt{(}\lfloor R \rfloor\texttt{)} \vdash M : \texttt{Bool}}{\Gamma \vdash \mathbf{delete}\ (x \texttt{<--}\, L)\ \mathbf{where}\, M : \texttt{()}}$$

$$\frac{}{\Gamma \vdash \cdot : \mathsf{ProvSpec}(R)} \qquad \frac{\Gamma \vdash S : \mathsf{ProvSpec}(R)}{\Gamma \vdash S, l\, \mathbf{prov}\, \mathbf{default} : \mathsf{ProvSpec}(R)}$$

$$\frac{\Gamma \vdash S : \mathsf{ProvSpec}(R) \qquad \Gamma \vdash M : \texttt{(}R\texttt{) -> (String,String,Int)}}{\Gamma \vdash S, l\, \mathbf{prov}\, M : \mathsf{ProvSpec}(R)}$$

Figure 3.1: Additional typing rules for Links^{W}.

$$\begin{aligned}
\lfloor O \rfloor &= O \\
\lfloor \mathbf{Prov}(A) \rfloor &= \lfloor A \rfloor \\
\lfloor (l_i : A_i)_{i=1}^{n} \rfloor &= (l_i : \lfloor A_i \rfloor)_{i=1}^{n} \\
R \triangleright \cdot &= R \\
(R, l : O) \triangleright (S, l\, \mathbf{prov}\, M) &= (R \triangleright S), l : \mathbf{Prov}(O)
\end{aligned}$$

Figure 3.2: Links^{W} type erasure and augmentation.

is S. As for ordinary tables, we check that the fields are of base type. These operations are defined in Figure 3.2. The database update rules make use of an *erasure* operation $\lfloor R \rfloor$ that takes a record or base type and replaces all occurrences of **Prov**(A) with A. We want to be able to update data but not provenance. Thus we erase the `Prov` type construcutor from record types when typechecking updates. The provenance specification on the table persists, so if we query an updated table, the new values will receive up-to-date provenance. Updating values in the database potentially renders existing annotations invalid. It might be possible to track annotated values and update or invalidate annotations when database updates are performed from inside of Links$^{\mathrm{W}}$. This is complicated by the fact that other clients can also write to the database. Links currently does not expose database transactions to the programmer so the same considerations already apply to unannotated values. For now, we restrict any guarantees to hold only in the absence of updates.

The following proofs and definitions are based on previous work by Buneman et al. [2008] in the context of nested relational algebra. The main correctness property of where-provenance is that annotations on values are propagated correctly. It should not be the case that we construct annotated values out of thin air. For the propagation behavior to be correct, it does not matter what the annotations are or where they come from. Buneman et al. discuss some other interesting properties which do not hold in our language. In their work, annotations are completely abstract, and queries have no way to inspect them. Therefore, they can show that queries are invariant under recoloring of the input. Links$^{\mathrm{W}}$ has the **prov** keyword to inspect provenance, therefore we cannot expect the same to hold here. However, we speculate that a similar property holds for sufficiently polymorphic functions by way of parametricity [Wadler, 1989]. In short, a polymorphic function cannot use **prov** to inspect provenance of its polymorphic arguments and only pass them on to other polymorphic functions. Therefore, it cannot possibly depend on the content of the annotations and is thus invariant under recoloring.

We assume a signature Σ where values inside tables are annotated with colors. We do not make any assumptions about these colors. However, they are particularly useful when they are distinct. In the case of distinct annotations on the input, we can look at the output and trace back annotated values to their source (assuming evaluation does not conjure up new annotated values out of

$$
\begin{array}{lcl}
cso_\Sigma(V^a) & = & \{V^a\} \cup cso_\Sigma(V) \\
cso_\Sigma(\mathbf{table}\, n) & = & cso_\Sigma(\Sigma(n)) \\
 & & \\
cso_\Sigma(c) & = & \emptyset \\
cso_\Sigma(\,[\,]\,) & = & \emptyset \\
cso_\Sigma(\,[M]\,) & = & cso_\Sigma(M) \\
cso_\Sigma(M +\!\!+ N) & = & cso_\Sigma(M) \cup cso_\Sigma(N) \\
cso_\Sigma((l_i = M_i)_{i=1}^n) & = & \bigcup_{i=1}^n cso_\Sigma(M_i) \\
cso_\Sigma(M.l) & = & cso_\Sigma(M) \\
cso_\Sigma(\mathbf{fun}\, f(x_i|_{i=1}^n)\, M) & = & cso_\Sigma(M) \\
cso_\Sigma(M(N_i|_{i=1}^n)) & = & cso_\Sigma(M) \cup \bigcup_{i=1}^n cso_\Sigma(N_i) \\
cso_\Sigma(\mathbf{var}\, x = M; N) & = & cso_\Sigma(M) \cup cso_\Sigma(N) \\
cso_\Sigma(\mathbf{if}\,(L)\, M\, \mathbf{else} N) & = & cso_\Sigma(L) \cup cso_\Sigma(M) \cup cso_\Sigma(N) \\
cso_\Sigma(\mathbf{query} M) & = & cso_\Sigma(M) \\
cso_\Sigma(\mathbf{empty}(M)) & = & cso_\Sigma(M) \\
cso_\Sigma(\mathbf{for}\,(x <\!- M)\, N) & = & cso_\Sigma(M) \cup cso_\Sigma(N) \\
cso_\Sigma(\mathbf{for}\,(x <\!\!-\!- M)\, N) & = & cso_\Sigma(M) \cup cso_\Sigma(N)
\end{array}
$$

Figure 3.3: Colored subobjects in LINKS^{W} expressions.

thin air). In Figure 3.3 we define the function cso_Σ for finding all *colored subobjects* of a LINKS^{W} value, and by extension, term. The interesting cases are for annotated values V^a and tables. Ultimately, we want to know that annotations on a query result point back to the database. The extension to terms allows us to find annotations in a program and state that we do not invent any during evaluation. Thus, if we start with a distinctly annotated database and no annotated constants, we can then guarantee that all annotated values in the result of evaluation come, without modification, directly from the database.

Theorem 3.5 formally states this intuition of evaluation not inventing annotated values. We first show a helpful lemma: the colored subobjects of a term substituted into an evaluation context $\mathcal{E}[M]$ can be obtained by considering the evaluation context $\mathcal{E}$ and term M separately, instead. We extend $cso_\Sigma(-)$ to operate on evaluation contexts in the obvious way.

Lemma 3.4. *Given evaluation context $\mathcal{E}$ and term M, we have:*

$$cso_\Sigma(\mathcal{E}[M]) = cso_\Sigma(\mathcal{E}) \cup cso_\Sigma(M)$$

Proof. Proof by induction on the structure of the evaluation context. In the case for $\mathcal{E} = []$ we take the colored subobjects of a hole to be the empty set. The other

cases are straightforward due to the compositional definition of $cso_\Sigma(-)$ and the properties of sets and set union $\cup$. □

Theorem 3.5 (Correctness of where-provenance)**.** *Let M and N be* Links^W *terms, and let Σ be a signature that provides annotated table rows. We have:*

$$\Sigma, M \longrightarrow \Sigma, N \Rightarrow cso_\Sigma(N) \subseteq cso_\Sigma(M)$$

Proof. Proof by induction on the derivation of the evaluation relation $\longrightarrow$.

- Case $(\mathbf{fun}\, f(x_i)\, M)(V_i) \longrightarrow M[f := \mathbf{fun}\, f(x_i)\, M, x_i := V_i]$:

$$\begin{aligned} cso_\Sigma(M[f := \mathbf{fun}\, f(x_i)\, M, x_i := V_i]) &\subseteq cso_\Sigma(M) \cup cso_\Sigma(\mathbf{fun}\, f(x_i)\, M) \cup \bigcup_{i=0}^{n} cso_\Sigma(V_i) \\ &= cso_\Sigma(\mathbf{fun}\, f(x_i)\, M) \cup \bigcup_{i=0}^{n} cso_\Sigma(V_i) \\ &= cso_\Sigma\left((\mathbf{fun}\, f(x_i)\, M)(V_i)\right) \end{aligned}$$

- Case $\mathbf{var}\, x = V; M \longrightarrow M[x := V]$:

$$cso_\Sigma(M[x := V]) \subseteq cso_\Sigma(M) \cup cso_\Sigma(V) = cso_\Sigma(\mathbf{var}\, x = V; M)$$

- Case $(l_i = V_i)_{i=1}^n.l_k \longrightarrow V_k$ where $1 \leq k \leq n$:

$$cso_\Sigma(V_k) \subseteq \bigcup_{i=1}^{n} cso_\Sigma(V_i) = cso_\Sigma((l_i = V_i)_{i=1}^n) = cso_\Sigma((l_i = V_i)_{i=1}^n.l_k)$$

- Case $\mathbf{if}\,(\mathbf{true})\, M\, \mathbf{else}\, N \longrightarrow M$:

$$cso_\Sigma(M) \subseteq cso_\Sigma(M) \cup cso_\Sigma(N) = cso_\Sigma(\mathbf{if}\,(\mathbf{true})\, M\, \mathbf{else}\, N)$$

- Case $\mathbf{if}\,(\mathbf{false})\, M\, \mathbf{else}\, N \longrightarrow N$:

$$cso_\Sigma(N) \subseteq cso_\Sigma(M) \cup cso_\Sigma(N) = cso_\Sigma(\mathbf{if}\,(\mathbf{false})\, M\, \mathbf{else}\, N)$$

- Case $\mathbf{query}\, M \longrightarrow M$: $cso_\Sigma(M) = cso_\Sigma(\mathbf{query}\, M)$
- Case $\mathbf{table}\, n \longrightarrow \Sigma(n)$: $cso_\Sigma(\Sigma(n)) = cso_\Sigma(\mathbf{table}\, n)$.
- Case $\mathbf{empty}(\,[\,]\,) \longrightarrow \mathbf{true}$: $cso_\Sigma(\mathbf{true}) = \emptyset = cso_\Sigma(\mathbf{empty}(\,[\,]\,))$

- Case $\mathbf{empty}(V) \longrightarrow \mathbf{false}$, where $V \neq [\,]$:

$$cso_{\Sigma}(\mathbf{false}) = \emptyset \subseteq cso_{\Sigma}(V) = cso_{\Sigma}(\mathbf{empty}(V))$$

- Case $\mathbf{for}\,(x \leftarrow [\,])\,M \longrightarrow [\,]$: $cso_{\Sigma}([\,]) = \emptyset \subseteq cso_{\Sigma}(\mathbf{for}\,(x \leftarrow [\,])\,M)$

- Case $\mathbf{for}\,(x \leftarrow [V])\,M \longrightarrow M[x := V]$:

$$cso_{\Sigma}(M[x := V]) \subseteq cso_{\Sigma}(M) \cup cso_{\Sigma}(V) = cso_{\Sigma}(\mathbf{for}\,(x \leftarrow [V])\,M)$$

- Case $\mathbf{for}\,(x \leftarrow V +\!\!+ W)\,M \longrightarrow (\mathbf{for}\,(x \leftarrow V)\,M) +\!\!+ (\mathbf{for}\,(x \leftarrow W)\,M)$:

$$\begin{aligned} cso_{\Sigma}(\mathbf{for}\,(x \leftarrow V +\!\!+ W)\,M) &= cso_{\Sigma}(V +\!\!+ W) \cup cso_{\Sigma}(M) \\ &= cso_{\Sigma}(V) \cup cso_{\Sigma}(W) \cup cso_{\Sigma}(M) \\ &= cso_{\Sigma}((\mathbf{for}\,(x \leftarrow V)\,M) +\!\!+ (\mathbf{for}\,(x \leftarrow W)\,M)) \end{aligned}$$

- Case $\mathbf{for}\,(x \mathrel{<\!\!-\!\!-} V)\,M \longrightarrow \mathbf{for}\,(x \leftarrow V)\,M$:

$$\begin{aligned} cso_{\Sigma}(\mathbf{for}\,(x \leftarrow V)\,M) &= cso_{\Sigma}(V) \cup cso_{\Sigma}(M) \\ &= cso_{\Sigma}(\mathbf{for}\,(x \mathrel{<\!\!-\!\!-} V)\,M) \end{aligned}$$

- Case $M \longrightarrow M' \Rightarrow \mathcal{E}[M] \longrightarrow \mathcal{E}[M']$ (evaluation step inside a context):

$$\begin{aligned} cso_{\Sigma}(\mathcal{E}[M']) &= cso_{\Sigma}(\mathcal{E}) \cup cso_{\Sigma}(M') && \text{Lemma 3.4} \\ &\subseteq cso_{\Sigma}(\mathcal{E}) \cup cso_{\Sigma}(M) && \text{IH} \\ &= cso_{\Sigma}(\mathcal{E}[M]) && \text{Lemma 3.4} \end{aligned}$$

□

In this section we made precise the static and dynamic semantics of Links^{W}, including the meaning of where-provenance annotations. Intuitively, given an annotated value, we can be sure that the same value is in the database, and was given the same initial annotation. If this initial annotation was produced by the **default** annotation strategy, we also know that the annotation points back to a database cell with the same value. Next, we discuss an implementation strategy for Links^{W} by type-directed source-to-source translation to plain Links.

$$
\begin{array}{rcl}
\mathfrak{W}[\![O]\!] & = & O \\
\mathfrak{W}[\![A \texttt{->} B]\!] & = & \mathfrak{W}[\![A]\!] \texttt{->} \mathfrak{W}[\![B]\!] \\
\mathfrak{W}[\![(l_i : A_i)_{i=1}^n]\!] & = & (l_i : \mathfrak{W}[\![A_i]\!])_{i=1}^n \\
\mathfrak{W}[\![\texttt{[}A\texttt{]}]\!] & = & \texttt{[}\mathfrak{W}[\![A]\!]\texttt{]} \\
\mathfrak{W}[\![\mathbf{Prov}(A)]\!] & = & (\mathsf{data} : \mathfrak{W}[\![A]\!], \mathsf{prov} : (\texttt{String}, \texttt{String}, \texttt{Int})) \\
\mathfrak{W}[\![\mathbf{table}(R)]\!] & = & (\mathbf{table}(\lfloor R \rfloor), () \texttt{->} \texttt{[}\mathfrak{W}[\![(R)]\!]\texttt{]})
\end{array}
$$

$$
\begin{array}{rcl}
\mathfrak{W}[\![\cdot]\!] & = & \cdot \\
\mathfrak{W}[\![\Gamma, x : A]\!] & = & \mathfrak{W}[\![\Gamma]\!], x : \mathfrak{W}[\![A]\!]
\end{array}
$$

Figure 3.6: Type translation for Links$^{\mathrm{W}}$.

3.3 Implementation

We define a type-directed translation from Links$^{\mathrm{W}}$ to Links based on the semantics presented in the previous section. The syntactic translation of types $\mathfrak{W}[\![A]\!]$ and contexts $\mathfrak{W}[\![\Gamma]\!]$ is shown in Figure 3.6. The expression translation function $\mathfrak{W}[\![M]\!]$ is shown in Figure 3.7. The cases up to table comprehensions **for** (<-- L) M are the obvious recursive application of $\mathfrak{W}$. The actual prototype implementation extends the Links parser and type checker, and desugars the new Links$^{\mathrm{W}}$ constructs to Links constructs after type checking, reusing the backend mostly unchanged.

Values of type **Prov**(O) are represented at runtime as ordinary Links records with type `(data:` O`, prov: (String, String, Int))`. The keywords **data** and **prov** translate to projections to the respective fields.

We translate table declarations to pairs. The first component is a simple table declaration where all columns have their primitive underlying non-provenance type. We will use this table declaration for insert, update, and delete operations. The second component is essentially a delayed query that calculates where-provenance for the entire table. The fact that it is delayed is important here, because it means that it can be inlined and simplified later, rather than loaded into memory. We compute provenance for each record by iterating over the underlying table. For every record of the input table, we construct a new record with the same fields as the table. For every column with provenance, the field's value is a record with `data` and `prov` fields. The `data` field is just the value.

$$
\begin{array}{rcl}
\mathfrak{W}[\![V^W]\!] & = & (\mathsf{data} = \mathfrak{W}[\![V]\!], \mathsf{prov} = \mathfrak{W}[\![W]\!]) \\
\mathfrak{W}[\![c]\!] & = & c \\
\mathfrak{W}[\![x]\!] & = & x \\
\mathfrak{W}[\![(l_i = M_i)_{i=1}^n]\!] & = & (l_i = \mathfrak{W}[\![M_i]\!])_{i=1}^n \\
\mathfrak{W}[\![N.l]\!] & = & \mathfrak{W}[\![N]\!].l \\
\mathfrak{W}[\![\mathbf{fun}(x_i|_{i=0}^n)\ \{M\}]\!] & = & \mathbf{fun}(x_i|_{i=0}^n)\ \{\mathfrak{W}[\![M]\!]\} \\
\mathfrak{W}[\![M(N_i|_{i=0}^n)]\!] & = & \mathfrak{W}[\![M]\!](\mathfrak{W}[\![N_i]\!]|_{i=0}^n) \\
\mathfrak{W}[\![\mathbf{var}\ x = M; N]\!] & = & \mathbf{var}\ x = \mathfrak{W}[\![M]\!]; \mathfrak{W}[\![N]\!] \\
\mathfrak{W}[\![\mathbf{query}\ \{M\}]\!] & = & \mathbf{query}\ \{\mathfrak{W}[\![M]\!]\} \\
\mathfrak{W}[\![\,[\,]\,]\!] & = & [\,] \\
\mathfrak{W}[\![\,[M]\,]\!] & = & [\mathfrak{W}[\![M]\!]] \\
\mathfrak{W}[\![M \mathbin{++} N]\!] & = & \mathfrak{W}[\![M]\!] \mathbin{++} \mathfrak{W}[\![N]\!] \\
\mathfrak{W}[\![\mathbf{if}\ (L)\ \{M\}\ \mathbf{else}\ \{N\}]\!] & = & \mathbf{if}\ (\mathfrak{W}[\![L]\!])\ \{\mathfrak{W}[\![M]\!]\}\ \mathbf{else}\ \{\mathfrak{W}[\![N]\!]\} \\
\mathfrak{W}[\![\mathbf{empty}\ (M)]\!] & = & \mathbf{empty}\ (\mathfrak{W}[\![M]\!]) \\
\mathfrak{W}[\![\mathbf{for}\ (x <- L)\ M]\!] & = & \mathbf{for}\ (x <- \mathfrak{W}[\![L]\!])\ \mathfrak{W}[\![M]\!] \\
\mathfrak{W}[\![\mathbf{where}(M)\ N]\!] & = & \mathbf{where}(\mathfrak{W}[\![M]\!])\ \mathfrak{W}[\![N]\!] \\
\mathfrak{W}[\![\mathbf{for}\ (x <-- L)\ M]\!] & = & \mathbf{for}\ (x <- \mathfrak{W}[\![L]\!].2())\ \mathfrak{W}[\![M]\!] \\
\mathfrak{W}[\![\mathbf{data}\ M]\!] & = & \mathfrak{W}[\![M]\!].\mathsf{data} \\
\mathfrak{W}[\![\mathbf{prov}\ M]\!] & = & \mathfrak{W}[\![M]\!].\mathsf{prov} \\
\mathfrak{W}[\![\mathbf{insert}\ L\ \mathbf{values}\ M]\!] & = & \mathbf{insert}\ \mathfrak{W}[\![L]\!].1\ \mathbf{values}\ \mathfrak{W}[\![M]\!] \\
\mathfrak{W}[\![\mathbf{update}\ (x <- L)\ \mathbf{where}\ M\ \mathbf{set}\ N]\!] & = & \mathbf{update}\ (x <- \mathfrak{W}[\![L]\!].1)\ \mathbf{where}\ \mathfrak{W}[\![M]\!]\ \mathbf{set}\ \mathfrak{W}[\![N]\!] \\
\mathfrak{W}[\![\mathbf{delete}\ (x <- L)\ \mathbf{where}\ M]\!] & = & \mathbf{delete}\ (x <- \mathfrak{W}[\![L]\!].1)\ \mathbf{where}\ \mathfrak{W}[\![M]\!] \\
\mathfrak{W}[\![\mathbf{table}\ n\ \mathbf{with}(R)\mathbf{where}\ S]\!] & = & \\
\multicolumn{3}{c}{(\mathbf{table}\ n\ \mathbf{with}\ (R),\ \ \mathbf{fun}()\{\mathbf{for}(x <-- \mathbf{table}\ n\ \mathbf{with}\ (R))\,[\ (R \triangleright_x^n S)\ ]\ \})}
\end{array}
$$

$$
\begin{array}{rcl}
\cdot \triangleright_x^n \cdot & = & \cdot \\
(R, l : O) \triangleright_x^n \cdot & = & (R \triangleright_x^n \cdot), l = x.l \\
(R, l : O) \triangleright_x^n (S, l\ \mathbf{prov}\ \mathbf{default}) & = & (R \triangleright_x^n S), l = (\mathbf{data} = x.l, \mathbf{prov} = (n, l_d, x.oid)) \\
(R, l : O) \triangleright_x^n (S, l\ \mathbf{prov}\ M) & = & (R \triangleright_x^n S), l = (\mathbf{data} = x.l, \mathbf{prov} = \mathfrak{W}[\![M]\!](x))
\end{array}
$$

Figure 3.7: Translation of Links^{W} to Links, and auxiliary operation $R \triangleright_x^n S$.

The `prov` field is a triple of the table's name, the column's name, and the row identifier as given by the `oid` column. The translation of table references also uses an auxiliary operation $R \triangleright_x^n S$ which, given a row type R, a table name n, a variable x and a provenance specification S, constructs a record in which each field contains data from x along with the specified provenance (if any). We wrap the iteration in an anonymous function to delay execution: otherwise, the provenance-annotated table would be constructed in memory when the table reference is first evaluated. We will eventually apply this function in a query, and the Links query normalizer will inline the provenance annotations and normalize them along with the rest of the query.

We translate table comprehensions to comprehensions over the second component of a translated table declaration. Since that component is a function, we have to apply it to a (unit) argument.

For example, recall the example database in Figure 2.3 on page 26 and consider the following Links^W table declaration with provenance annotations on the phone number column:

```
var agencies =
  table "Agencies"
  with (name: String, based_in: String, phone: String)
  where phone prov default
```

This would translate to the following pair of a plain Links table declaration for use in updates and a delayed query fragment that provides initial annotations:

```
var agencies =
  (table "Agencies"
   with (name: String, based_in: String, phone: String),
   fun () { for (t <-- table "Agencies"
                       with (name: String, based_in: String,
                             phone: String))
               [(name = t.name, based_in = t.based_in,
                 phone = (data = t.phone,
                          prov = ("Agencies", "phone", t.oid)))] })
```

We assume a second table declaration that is similiarly translated:

```
var externalTours =
  table "ExternalTours"
  with (name:String, destination:String, type:String, price:Int)
  where destination prov default, price prov default
```

Now we can write the following variant of the boat tours query from Figure 2.4 that returns the agency's phone number's where-provenance in a separate column `p_phone`:

```
for (a <-- agencies)
  for (e <-- externalTours)
  where (a.name == e.name && e.type == "boat")
    [(name = e.name,
      phone = data a.phone,
      p_phone = prov a.phone)]
```

The translated query looks like this:

```
for (a <-- agencies.2())
  for (e <-- externalTours.2())
  where (a.name == e.name && e.type == "boat")
    [(name = e.name,
      phone = a.phone.data,
      p_phone = a.phone.prov)]
```

Query normalization will inline the references to the variables `agencies` and `externalTours` and the resulting expression is too big to reasonably print. However, query normalization will also symbolically execute projections like `agencies.2`, inline the delayed query fragements, remove unused expressions, and ultimately generate the following SQL query.

```
SELECT e.name AS name,
       a.phone AS phone,
       'agencies' AS p_phone_1,
       'phone' AS p_phone_2,
       a.oid AS p_phone_3
  FROM agencies AS a, externaltours AS e
 WHERE a.name = e.name AND e.type = 'boat'
```

In this query, the table and column part of the where-provenance are in fact static, and the generated SQL query reflects this by using constants in the select clause. We see no trace of pairing a table with a view for initial annotations, function applications, or nested record projections to `data` or `prov` fields.

The type-preservation correctness property of the where-provenance translation is that it preserves well-formedness. We first need

Lemma 3.8. *Let R be a row and S be a provenance specification. Then*

- $\mathfrak{W}[\![(\lfloor R \rfloor)]\!] = (R)$,

- $\lfloor\!\lfloor (R \triangleright S) \rfloor\!\rfloor = (R)$.

The type-preservation property for the translation is stated as follows.

Theorem 3.9.

1. *For every* Links$^{\mathrm{W}}$ *context* Γ, *term* M, *and type* A, *if* $\Gamma \vdash_{\text{Links}^{\mathrm{W}}} M : A$ *then* $\mathfrak{W}[\![\Gamma]\!] \vdash_{\text{Links}} \mathfrak{W}[\![M]\!] : \mathfrak{W}[\![A]\!]$.

2. *For every* Links$^{\mathrm{W}}$ *context* Γ, *provenance specification* S, *row* R *and subrow* R' *such that* $R' \triangleright^n_x S$ *is defined, if* $\Gamma \vdash S : \mathsf{ProvSpec}(R)$ *then* $\mathfrak{W}[\![\Gamma]\!], x\!:(R) \vdash (R' \triangleright^n_x S) : \mathfrak{W}[\![(R' \triangleright S)]\!]$.

Proof. Proof is by induction on the structure of Links$^{\mathrm{W}}$ derivations. Most cases for the first part are immediate; we show some representative examples.

- If the derivation is of the form

$$\frac{\Gamma \vdash M : \mathbf{Prov}(A)}{\Gamma \vdash \mathbf{data}\, M : A}$$

  then by induction we have $\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!] : \mathfrak{W}[\![\mathbf{Prov}(A)]\!]$, and can conclude:

$$\frac{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!] : (\mathsf{data} : \mathfrak{W}[\![A]\!], \mathsf{prov} : (\texttt{String}, \texttt{String}, \texttt{Int}))}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!].\mathsf{data} : \mathfrak{W}[\![A]\!]}$$

- If the derivation is of the form

$$\textsc{Data}\ \frac{\Gamma \vdash M : \mathbf{Prov}(A)}{\Gamma \vdash \mathbf{prov}\, M : \texttt{(String,String,Int)}}$$

  then by induction we have $\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!] : \mathfrak{W}[\![\mathbf{Prov}(A)]\!]$, and can conclude:

$$\frac{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!] : (\mathsf{data} : \mathfrak{W}[\![A]\!], \mathsf{prov} : (\texttt{String}, \texttt{String}, \texttt{Int}))}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!].\mathsf{prov} : \texttt{(String,String,Int)}}$$

- If the derivation is of the form

$$\textsc{Table}\ \frac{R :: \mathsf{BaseRow} \qquad \Gamma \vdash S : \mathsf{ProvSpec}(R)}{\Gamma \vdash \mathbf{table}\, n\, \mathbf{with}\, (R)\, \mathbf{where}\, S : \mathbf{table}\, (R \triangleright S)}$$

Then since $\|R \triangleright S\| = R$ (Lemma 3.8) we can conclude:

$$\frac{}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{table}\ n\ \mathbf{with}\ (R) : \mathbf{table}(\|R \triangleright S\|)}$$

and by the second induction hypothesis,

$$\frac{\dfrac{\dfrac{R :: \mathsf{BaseRow}}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{table}\ n\ \mathbf{with}\ (R) : \mathbf{table}(R)} \quad \dfrac{\mathfrak{W}[\![\Gamma]\!], x\!:\!(R) \vdash (R \triangleright^{n}_{x} S) : \mathfrak{W}[\![(R \triangleright S)]\!]}{\mathfrak{W}[\![\Gamma]\!], x\!:\!(R) \vdash [(R \triangleright^{n}_{x} S)] : [\mathfrak{W}[\![(R \triangleright S)]\!]]}}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{for}(x \leftarrow\!\!- \mathbf{table}\ n\ \mathbf{with}\ (R))\,[(R \triangleright^{n}_{x} S)] : [\mathfrak{W}[\![(R \triangleright S)]\!]]}}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{fun}()\{\mathbf{for}(x \leftarrow\!\!- \mathbf{table}\ n\ \mathbf{with}\ (R))\,[(R \triangleright^{n}_{x} S)]\} : () \to [\mathfrak{W}[\![(R \triangleright S)]\!]]}$$

- If the derivation is of the form

$$\frac{\textsc{For-Table}\quad \Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma, x : (R) \vdash M : [B]}{\Gamma \vdash \mathbf{for}\ (x \leftarrow\!\!- L)\ M : [B]}$$

  then by induction we have $\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!] : (\mathbf{table}(\|R\|), () \to [\mathfrak{W}[\![(R)]\!]])$, so we can proceed as follows:

$$\frac{\dfrac{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!].2 : () \to [\mathfrak{W}[\![(R)]\!]]}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!].2() : [\mathfrak{W}[\![(R)]\!]]} \qquad \mathfrak{W}[\![\Gamma]\!], x : \mathfrak{W}[\![(R)]\!] \vdash \mathfrak{W}[\![M]\!] : [\mathfrak{W}[\![B]\!]]}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{for}\ (x \leftarrow \mathfrak{W}[\![L]\!].2())\ \mathfrak{W}[\![M]\!] : [\mathfrak{W}[\![B]\!]]}$$

- If the derivation is of the form

$$\frac{\textsc{Delete}\quad \Gamma \vdash L : \mathbf{table}(R) \qquad \Gamma, x : (\|R\|) \vdash M : \mathtt{Bool}}{\Gamma \vdash \mathbf{delete}\ (x \leftarrow\!\!- L)\ \mathbf{where}\ M : ()}$$

  then by induction we have $\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!] : \mathfrak{W}[\![\mathbf{table}(R)]\!]$ and $\mathfrak{W}[\![\Gamma]\!], x : \mathfrak{W}[\![(\|R\|)]\!] \vdash \mathfrak{W}[\![M]\!] : \mathtt{Bool}$.

$$\frac{\dfrac{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!] : (\mathbf{table}(\|R\|), () \to [(R)])}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![L]\!].1 : \mathbf{table}(\|R\|)} \qquad \mathfrak{W}[\![\Gamma]\!], x : (\|R\|) \vdash \mathfrak{W}[\![M]\!] : \mathtt{Bool}}{\mathfrak{W}[\![\Gamma]\!] \vdash \mathbf{delete}\ (x \leftarrow\!\!- \mathfrak{W}[\![L]\!].1)\ \mathbf{where}\ \mathfrak{W}[\![M]\!] : ()}$$

For the second part, we proceed by induction on the structure of the derivation of $\Gamma \vdash S : \mathsf{ProvSpec}(R)$. We show one representative case, for derivations of the form

$$\frac{\Gamma \vdash S : \mathsf{ProvSpec}(R) \qquad \Gamma \vdash M : (R) \to (\mathtt{String}, \mathtt{String}, \mathtt{Int})}{\Gamma \vdash S, l\ \mathbf{prov}\ M : \mathsf{ProvSpec}(R)}$$

In this case, by induction we have that $\mathfrak{W}[\![\Gamma]\!], x\colon (R) \vdash (R' \triangleright_x^n S) : \mathfrak{W}[\![(R' \triangleright S)]\!]$ holds for any subrow R' of R, and by the first induction hypothesis we also know that $\mathfrak{W}[\![\Gamma]\!] \vdash \mathfrak{W}[\![M]\!] : \mathfrak{W}[\![(R)]\!]$ `-> (String,String,Int).`

Suppose $R', l : O \triangleright_x^n S, l$ **prov** M. Then we can conclude that

$$\mathfrak{W}[\![\Gamma]\!], x\colon (R) \vdash (R', l : \mathbf{Prov}(O) \triangleright_x^n S, l \,\mathbf{prov}\, M) : \mathfrak{W}[\![(R', l : O \triangleright S, l \,\mathbf{prov}\, O)]\!]$$

because $(R', l : O \triangleright_x^n S, l \,\mathbf{prov}\, M) = (R' \triangleright_x^n S), l = (\mathbf{data} = x.l, \mathbf{prov} = \mathfrak{W}[\![M]\!](x))$ and $R', l : O \triangleright S, l \,\mathbf{prov}\, O = (R' \triangleright S), l : \mathbf{Prov}(O)$. □

Theorem 3.10. *For all* Links$^{\mathrm{W}}$ *terms* M, M' *and signatures* Σ, Σ'*, if* $\Sigma, M \longrightarrow \Sigma', M'$*, then* $\mathfrak{W}[\![\Sigma]\!], \mathfrak{W}[\![M]\!] \longrightarrow^* \mathfrak{W}[\![\Sigma']\!], \mathfrak{W}[\![M']\!]$.

Proof. By induction on the evaluation relation $\longrightarrow$. We show the most interesting cases.

- **for** $(x$ `<--` **table** t **with** R **where** $S)$ $M \longrightarrow$ **for**$(x$ `<--` $\Sigma(t))$ M: The left hand side $\mathfrak{W}[\![$**for** $(x$ `<--` **table** t **with** R **where** $S)$ $M]\!]$ translates to

  ```
  for (x <- (table t with R, fun () {
                for (x <-- table t with R)
                  [(R ▷ₓⁿ S)]
                }).2())
    𝔚⟦M⟧
  ```

  which evaluates in multiple steps to:

  ```
  for (x <- for (x <-- table t with R) [(R ▷ₓⁿ S)]) 𝔚⟦M⟧
  ```

  The inner comprehension is just the Links computation of initial annotations and is semantically equivalent to looking up an annotated table in the signature Σ, and therefore the translation of the right-hand side.

- **prov** $M \longrightarrow$ **prov** M': We have $\mathfrak{W}[\![\mathbf{prov} M]\!] = \mathfrak{W}[\![M]\!].\mathsf{prov}$ and $\mathfrak{W}[\![\mathbf{prov} M']\!] = \mathfrak{W}[\![M']\!].\mathsf{prov}$. By IH, $\mathfrak{W}[\![M]\!].\mathsf{prov} \longrightarrow^* \mathfrak{W}[\![M']\!].\mathsf{prov}$.

- **prov** $V^W \longrightarrow W$: We have $\mathfrak{W}[\![\mathbf{prov}\, V^W]\!] = (\mathsf{data} = \mathfrak{W}[\![V]\!], \mathsf{prov} = \mathfrak{W}[\![W]\!]).\mathsf{prov}$ which evaluates to $\mathfrak{W}[\![W]\!]$. □

We have shown that annotation-propagation in Links$^{\mathrm{W}}$ is color-propagating (Theorem 3.5 on page 43), that the translation to Links is type-preserving (Theorem 3.9), and that the implementation by translation to plain Links simulates the semantics (Theorem 3.10).

3.4 Discussion

In this chapter we presented the design and implementation of Links^W — a programming language with support for where-provenance. It features fine grained control over which values carry annotations. Annotated values have separate types from unannotated values and the type system guarantees that annotations are accurate. This prevents programmers from accidentally modifying or erasing annotations.

There is limited support for custom annotations. For each cell, Links^W can run a user-defined function that takes the row as input and produces a provenance triple. This function can be used to extract external annotations stored in the same row, but is not quite flexible enough for using external annotations stored in a separate table. Imagine annotations for table *as*, column *c* were stored in table *ps* column c_t, c_c, and c_r with rows identified by a common column *i*. We would want to write a function like the following to generate provenance for $as.c$ by querying $ps.c_*$ like this:

```
fun (a) {
  for (p <-- table ps where ...)
  where (a.i == p.i)
    [(p.c_t, p.c_c, p.c_r)] }
```

However, this function has the wrong type: it produces a list of annotations. There is a function `the : ([a]) ~> a` to extract the element of a singleton list, but it is partial because the list might be empty or contain more than one (different) element and thus cannot be used in queries. Links currently lacks a mechanism for declaring uniqueness and foreign key constraints that would make such queries safe. We could have made where-provenance-annotations lists of triples instead of exactly one triple to partly avoid this problem. However, that would require support for nested collections instead of only record flattening. Links supports nested collections in queries [Cheney et al., 2014c], but such queries tend to be more expensive. It is also not clear how we could enforce non-empty lists of annotations.

We limit annotations themselves to be of type `(String,String,Int)`. This is overly and unnecessarily restrictive. We could parameterize the `Prov` type by the type of the annotation in addition to the type of the data. Similarly, we limit the data to be of base type. Buneman et al. [2008] consider where-

provenance in a nested data model where values at any level can be annotated. We could likely lift the cell-level restriction in Links^W and support annotations on whole rows and tables.

The implementation of Links^W by translation to plain Links queries is quite naive. However, thanks to query normalization, it does not necessarily lead to bad queries and the compositionality offers the benefit that where-provenance can be used in the same query that generates it. For example, we can use where-provenance to filter data based on its source table. Where-provenance is often at least in parts static and normalization exposes this without any special-purpose static analysis. This can lead to parts of a source query never actually appearing in the Sql translation. Even when filtering has to happen on dynamic data, we expect it to be usually much better to push filtering based on provenance down into the query rather than to produce the whole result with provenance annotations and then filter the annotated result afterwards.

One area that has not received much attention in practical implementations of provenance is database updates. Links^W is no exception here: the correctness properties assume that the underlying database is unchanging. This is of course not a realistic assumption: Links includes update operations that can change the database tables, and other database users might concurrently update the data or even change the structure of the data.

An important design decision in Links^W was to have fine grained where-provenance annotations by changing table definitions to indicate which columns carry provenance. The type system then forces all users of such tables to adapt accordingly, by either propagating annotations or discarding them where necessary. We consider the manual annotation effort reasonable because queries are typically small. We explore one alternative, automatically adding annotations everywhere, in Links^L in the next chapter.

Provenance polymorphism could also reduce annotation effort. Some functions, like the identity, are sufficiently polymorphic to work both on annotated and unannotated data. Other functions, such as addition clearly produce unannotated data. However, there are functions that fall in between these two extremes. Consider the binary `min` function that returns the smaller of its arguments for example. We can write a version for plain `Int`:

```
sig min : (Int, Int) -> Int
fun min (a, b) { if (a < b) { a } else { b } }
```

and a version for annotated `Prov(Int)`:

```
sig min' : (Prov(Int), Prov(Int)) -> Prov(Int)
fun min' (a, b) { if (data a < data b) { a } else { b } }
```

Currently it is not possible in LINKS^{W} to write a single function definition that covers both cases. Some form of subtyping or provenance polymorphism could be an interesting extension. In other languages, provenance-annotated values can and should be good citizens and participate in existing abstractions such as type classes.

Chapter 4

Links^{L} — lineage in Links

This chapter includes material from previously published work [Fehrenbach and Cheney, 2016, 2018].

This chapter describes a programming language called Links^{L} with built-in support for lineage. Section 4.1 gives a brief introduction to the language by way of examples and discusses some aspects of language design. Section 4.2 describes syntax and semantics, including a precise definition of lineage. Section 4.3 describes a type-directed source-to-source translation from Links^{L} to plain Links as one possible implementation strategy. In Section 4.4 we discuss related work on future extensions. We evaluate the performance of a prototype implementation in Chapter 5.

4.1 Overview

Links^{L} only includes one new keyword compared to plain Links: **`lineage`** indicates that a query should return a result where each row is annotated with lineage. For example, we can request the lineage of the boat tours query from Figure 2.4 on page 27 like this:

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

This query produces the following result given the example database from Figure 2.3 on page 26. Compared to the plain result, each row is annotated with

name	phone	lineage
EdinTours	412 1200	[(Agencies,1),(ExternalTours,5)]
EdinTours	412 1200	[(Agencies,1),(ExternalTours,6)]
Burns's	607 3000	[(Agencies,2),(ExternalTours,7)]

a list of pairs of table name and row number, which represent all of the rows that were "necessary" to produce the result row.[1] For example, the last row would not be in the result if either the second row of the agencies table, or the row with `oid` 7 of the external tours table, were missing. The multiset of annotations is a safe over-approximation of the set of rows that is strictly necessary to produce the result row — any row that is not in the annotations could not have been necessary. We give a precise definition of the meaning of lineage in Links^{L} in Section 4.2, Theorem 4.12.

We have taken some liberties in the presentation as a table. The actual result is a Links^{L} value with the following type:

```
[(data: (name: String, phone: String)
  prov: [(String, Int)])]
```

Compare this to the type of the original query:

```
[(name: String, phone: String)]
```

The list type in the original query result type has been replaced by a pair of the original data and a list of lineage annotations, which are pairs of table name and row number.

The example also demonstrates how Links^{L} treats multiset results. The data portion of a lineage-annotated result is the same as it would have been with a plain, unannotated query. Links^{L} annotates each row independently, rather than merging annotations for equal data. This is why we have two EdinTours results — the original query also had two EdinTours results. Inspecting the lineage annotations and referring back to the tables, we can identify the first result as the Loch Ness tour and the second as the Firth of Forth tour.

The **`lineage`** keyword instructs Links^{L} to add annotations to every row in a query result, including in nested lists. For example, consider the following

[1] Again, we use PostgreSQL's automatically generated `oid` column for row numbers.

query that returns the names of all presidents together with the dates of their inaugurations that saw 193000 or more Metro customers before 11 a.m.

```
query {
  for (p <-- presidents)
    [(name = p.name,
      dates = for (i <-- inaugurations) where (i.nth == p.nth)
              for (m <-- metro) where (m.date == i.date &&
                     m.time == 11 && m.trips >= 193000)
                [i.date])] }
```

When run against the database in Figure 1.1 on page 2 (on only the rows shown), this produces the following result:

```
[(dates = ["1/21/2013", "1/20/2009"], name = "Barack Obama"),
 (dates = ["1/20/2017"], name = "Donald Trump"),
 (dates = [], name = "George Washington")]
```

Believers in so-called "alternative facts"[2] might expect the lists of dates for presidents other than Trump to be empty. If we change **query** to **lineage** in the above query we get an annotated result (see Figure 4.1) that explains *why* every bit of data in the result is there. Note that we again use the `oid` column, which is automatically populated by PostgreSQL, as row identifiers. The lineage annotations on the outer list point to rows in the presidents table. Since the query does not filter presidents, each president in the table generates one row in the output. The annotations on each presidents inauguration dates point to a row in the inaugurations table and a row in the metro table. Together with the row of the presidents table, these fulfill all of the join and selection criteria and are thus sufficient for the date to be in the result.

Besides the different form of provenance, Links^{W} and Links^{L} explore slightly different areas of the design space for language-integrated provenance. In Links^{W} the programmer specifies exactly which values carry where-provenance annotations. In Links^{L} the programmer requests lineage annotations for a whole query and the language adds them to every list type. Unlike where-provenance-annotated values in Links^{W}, lineage-annotated values in Links^{L} are plain data — not elements of some abstract type. Thus Links^{W} offers more precise control over what gets annotated and greater type-safety when operating with annotated values. However, annotating everything requires more effort. More interestingly, even annotating only a part of the result can result in effort

[2] Alternative facts are more commonly known as lies.

```
[(data = (dates = [(data = "1/20/2009",
                    prov = [(row = 26250334, table = "inaugurations"),
                            (row = 26250338, table = "metro")]),
                   (data = "1/21/2013",
                    prov = [(row = 26250335, table = "inaugurations"),
                            (row = 26250337, table = "metro")])],
          name = "Barack Obama"),
  prov = [(row = 26250331, table = "presidents")]),
 (data = (dates = [(data = "1/20/2017",
                    prov = [(row = 26250336, table = "inaugurations"),
                            (row = 26250340, table = "metro")])],
          name = "Donald Trump"),
  prov = [(row = 26250332, table = "presidents")]),
 (data = (dates = [], name = "George Washington"),
  prov = [(row = 26250330, table = "presidents")])]
```

Figure 4.1: Lineage-annotated inauguration dates result.

proportional to the size of the query. Expressions and functions which operate on where-provenance-annotated data may need to be adapted to make them type-check in LINKS^{W}. In contrast, changing **`query`** to **`lineage`** causes LINKS^{L} to annotate everything automatically. The programmer only has to adapt their surrounding program, if any, to deal with the annotations. LINKS^{L} will rewrite the query to propagate lineage, even if the query contains functions.

4.2 Syntax & semantics

On the surface, LINKS^{L} only adds the keyword **`lineage`** to the syntax of plain LINKS as defined in Figure 2.5 on page 31.

$$L, M, N \quad ::= \quad \cdots \mid \textbf{lineage}\{M\}$$

Like the keyword **`query`**, **`lineage`** is followed by a block of code that will be translated into SQL and executed on the database. The **`query`** keyword only affects where and how the evaluation takes place; the result is the same as if database tables were lists in memory. The **`lineage`** keyword causes the query to be rewritten to not only compute the result, but also annotate every row of the result with its lineage.

$$
\begin{array}{rcl}
\mathbf{Lin}(A) & = & (\mathsf{data} : A, \mathsf{prov} : \texttt{[(String,Int)]}) \\
\mathfrak{L}[\![O]\!] & = & O \\
\mathfrak{L}[\![A \texttt{->} B]\!] & = & \mathfrak{L}[\![A]\!] \texttt{->} \mathfrak{L}[\![B]\!] \\
\mathfrak{L}[\![(l_i : A_i)^{\,n}_{i=1}]\!] & = & (l_i : \mathfrak{L}[\![A_i]\!])^{\,n}_{i=1} \\
\mathfrak{L}[\![\texttt{[}A\texttt{]}]\!] & = & \texttt{[}\mathbf{Lin}(\mathfrak{L}[\![A]\!])\texttt{]} \\
\mathfrak{L}[\![\mathbf{table}(R)]\!] & = & \mathfrak{L}[\![\texttt{[} (R) \texttt{]}]\!]
\end{array}
$$

Figure 4.2: Lineage type translation.

$$
\textsc{Lineage}\ \frac{\Gamma \vdash M : \texttt{[}A\texttt{]} \qquad A :: \mathsf{QType}}{\Gamma \vdash \mathbf{lineage}\ \{M\} : \mathfrak{L}[\![\texttt{[}A\texttt{]}]\!]}
$$

Figure 4.3: Additional typing rule for Links$^{\mathsf{L}}$ (see Figure 2.7 for Links).

If M has type $\texttt{[}A\texttt{]}$ (which must be an appropriate query result type) then the type of $\mathbf{lineage}\,\{M\}$ is $\mathfrak{L}[\![\texttt{[}A\texttt{]}]\!]$, where $\mathfrak{L}[\![-]\!]$ is a type translation that adjusts the types of collections $\texttt{[}A\texttt{]}$ to allow for lineage, as shown in Figures 4.2 and 4.3.

Conceptually, a **lineage** block evaluates in one step to its result, as can be seen in Figure 4.4. The result is determined by a second evaluation relation that is only used "inside" lineage blocks: $\longrightarrow_{\mathsf{L}}$. The language which $\longrightarrow_{\mathsf{L}}$ operates on is Links$^{\mathsf{L}}$, except that list values are replaced by a variant of lists, $\hat{L}$, where every list element is annotated with a set of colors:

$$
\begin{array}{rcl}
V & ::= & \cdots \mid \hat{L} \\
\hat{L} & ::= & \texttt{[]} \mid \texttt{[}V\texttt{]}^{\,a} \mid \hat{L} +\!\!+ \hat{L} \\
M & ::= & \cdots \mid M^{\cup b}
\end{array}
$$

Note how the set of annotations a is on the singleton list constructor, not the actual element value as you might expect. We use annotations to track lineage, which describes *why* the value, or row, is in the result. Lineage is not concerned with what the value actually is.

We consider lineage to be a list of rows in the database and identify them by their table name and row number, rather than their contents. The type translation $\mathfrak{L}[\![-]\!]$ replaces every occurrence of the list type constructor in the type of a lineage query result by a list of records of data and its provenance. For

$$\frac{\hat{\Sigma}, \mathit{annotate}(M) \longrightarrow_{\mathsf{L}}^{*} \hat{\Sigma}, \hat{L}}{\Sigma, \mathbf{lineage}\, M \longrightarrow \Sigma, \mathit{a2d}(\hat{L})}$$

$$\begin{aligned}
\mathit{annotate}(\texttt{[]}) &= \texttt{[]} \\
\mathit{annotate}(\texttt{[}V\texttt{]}) &= \texttt{[}\mathit{annotate}(V)\texttt{]}^{\emptyset} \\
\mathit{annotate}(V +\!\!+ W) &= \mathit{annotate}(V) +\!\!+ \mathit{annotate}(W)
\end{aligned}$$

$$\begin{aligned}
\mathit{a2d}(\texttt{[]}) &= \texttt{[]} \\
\mathit{a2d}(\texttt{[}V\texttt{]}^{\{a_1,\ldots,a_n\}}) &= \texttt{[(data} = \mathit{a2d}(V), \texttt{prov} = \texttt{[}a_1, \ldots, a_n\texttt{])]} \\
\mathit{a2d}(V +\!\!+ W) &= \mathit{a2d}(V) +\!\!+ \mathit{a2d}(W)
\end{aligned}$$

Figure 4.4: Additional Links^{L} evaluation rule and helper functions.

example, if a **query** block has type `[Bool]`, the result of the same code in a **lineage** block has type `[(data: Bool, prov: [(String, Int)])]`.

The evaluation rule for the **lineage** keyword uses two helper functions, also defined in Figure 4.4, for going from Links^{L} values to annotated values used inside **lineage** blocks, and back. The first function is *annotate*, which recursively annotates Links^{L} lists with empty lineage annotations. We assume an extension of this function to non-list values and arbitrary Links^{L} terms in the obvious way. Only rows in database tables will have nonempty lineage annotations, provided by an extended context $\hat{\Sigma}$. The second function is *a2d*, which recursively transforms annotated lists into plain data Links^{L} lists. Non-list values are traversed in the obvious way. Every annotated list element will be transformed into a record with `data` and `prov` fields. The `prov` field will hold the lineage annotations as a list. To match the typing rule, annotations have to be pairs of a string and a number. In the prototype, we use the table name and row number, but it is conceivable to generalize lineage annotations in the future.

As Figure 4.5 on the facing page shows, evaluation inside **lineage** blocks is almost the same as evaluation outside. The interesting rules are those for adding annotations to singleton lists and the singleton comprehension rule. A **lineage** block is similar to a **query** block in that it can contain only pure, non-recursive functions, and no database updates. We do not support `empty` inside lineage blocks, because it can lead to non-monotonic queries (see Section 2.1.1.3). The major differences from ordinary evaluation are in the treatment of **for**

$$
\begin{aligned}
\hat{\Sigma}, [\,]^{\cup b} &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, [\,] \\
\hat{\Sigma}, ([V]^{a})^{\cup b} &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, [V]^{a \cup b} \\
\hat{\Sigma}, (V +\!\!+ W)^{\cup b} &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, V^{\cup b} +\!\!+ W^{\cup b} \\
\hat{\Sigma}, (\mathbf{fun}\, f(x_i|_{i=0}^{n})\, M)(V_i|_{i=0}^{n}) &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, M[x_i := V_i]_{i=0}^{n} \\
\hat{\Sigma}, \mathbf{var}\, x = V; M &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, M[x := V] \\
\hat{\Sigma}, \mathbf{for}\,(x <\!\!- \;\; [\,])\, M &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, [\,] \\
\hat{\Sigma}, \mathbf{for}\,(x <\!\!- [V]^{a})\, M &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, (M[x := V])^{\cup a} \\
\hat{\Sigma}, \mathbf{for}\,(x <\!\!- V +\!\!+ W)\, M &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, (\mathbf{for}\,(x <\!\!- V)\, M) +\!\!+ \mathbf{for}\,(x <\!\!- W)\, M \\
\hat{\Sigma}, \mathbf{for}\,(x <\!\!- \mathbf{table}\, t)\, M &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, \mathbf{for}\,(x <\!\!- \hat{\Sigma}(t))\, M \\
\hat{\Sigma}, \mathbf{query}(V) &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, V \\
\hat{\Sigma}, \mathbf{if}(\mathbf{true})\, M\, \mathbf{else}\, N &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, M \\
\hat{\Sigma}, \mathbf{if}(\mathbf{false})\, M\, \mathbf{else}\, N &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, N \\
\hat{\Sigma}, (l_i = V_i)_{i=1}^{n}.l_k &\longrightarrow_{\mathsf{L}} \hat{\Sigma}, V_k
\end{aligned}
$$

$$
\mathcal{E} \quad ::= \quad \cdots \mid \mathcal{E}^{\cup b}
$$

Figure 4.5: Propagation of lineage annotations.

comprehensions and the new syntax $M^{\cup b}$. A table comprehension takes the table values from an annotated signature $\hat{\Sigma}$, which maps tables to lists with lineage annotations. A **for** comprehension over a singleton list adds the singleton's annotation to all of the elements in the output list. For this use alone we introduce the new type of expression $M^{\cup b}$. It takes a term M and a set of annotations b, evaluates the term to a list value, and adds the annotations to any existing annotations. This is not syntax intended to be used by the programmer.

Lineage of a query result tells us which elements of the input were responsible for each element of the output to exist. If we run the same query again, but on only that part of the input that was mentioned in the lineage annotations, we should get (at least) the same output. (Note that the lineage annotations that Links$^{\mathsf{L}}$ produces are an over-approximation of the strictly necessary input rows, but still usually much better than annotating every row with the whole database.) In order to state this correctness property formally, we need three auxiliary definitions. The function $\|\cdot\|$ collects all lineage annotations mentioned in a value and, by straightforward extension, Links$^{\mathsf{L}}$ term. It is defined in Figure 4.6.

$$
\begin{aligned}
\|\,[M]^{a}\| &= a \cup \|M\| \\
\|\,[\,]\,\| &= \emptyset \\
\|M +\!\!+ N\| &= \|M\| \cup \|N\| \\
\|M^{\cup b}\| &= b \cup \|M\| \\
\|\mathtt{table}\, t\| &= \|\hat{\Sigma}(t)\| \\
\|\mathtt{var}\, x = M; N\| &= \|M\| \cup \|N\| \\
\|c\| &= c \\
\|(l_i = M_i)_{i=1}^{n}\| &= \bigcup_{i=1}^{n} \|M_i\| \\
\|M.l\| &= \|M\| \\
\|\mathtt{fun}\, f(x_i|_{i=1}^{n})\, M\| &= \mathtt{fun}\, f(x_i|_{i=1}^{n})\, \|M\| \\
\|\mathtt{if}\,(L)\, M\, \mathtt{else}\, N\| &= \|L\| \cup \|M\| \cup \|N\| \\
\|\mathtt{query}\, M\| &= \|M\| \\
\|\mathtt{for}\,(x <\!- M)\, N\| &= \|M\| \cup \|N\| \\
\|\mathtt{for}\,(x <\!- M)\, N\| &= \|M\| \cup \|N\|
\end{aligned}
$$

Figure 4.6: Collecting lineage annotations from an expression.

The function $\cdot|_b$, defined in Figure 4.7, restricts values, in particular list elements, to those annotated with a subset of annotations b. We extend this to Links$^{\mathrm{L}}$ terms in the obvious way and to annotated contexts such that tables mentioned in a restricted context $\hat{\Sigma}|_b$ do not contain rows which are not in b. Note that this function always preserves list literals and values originating in the surrounding program because those are annotated with empty lineage. The subset relationship in the case distinctions in restriction is perhaps not in the direction one would intuitively expect. For example, if we have $([M]^{\{x_1\}})^{\cup\{x_2\}}$ and restrict to $\{x_1, x_2\}$, then the $\{x_2\} \subseteq \{x_1, x_2\}$ case applies and we ultimately have $([M|_{\{x_1,x_2\}}]^{\{x_1\}})^{\cup\{x_2\}}$. This matches the behavior on values, i.e., $([M]^{\{x_1,x_2\}})|_{\{x_1,x_2\}}$ is $[M|_{\{x_1,x_2\}}]^{\{x_1,x_2\}}$.

Finally we have the recursive sublist relation $\sqsubseteq$, defined in Figure 4.8. For example, `[(a = [2])]` $\sqsubseteq$ `[(a = [1]), (a = [2, 3])]`.

Suppose a monotonic Links$^{\mathrm{L}}$ query q evaluates, inside a lineage block, to an annotated value $\hat{v}$ in a context $\hat{\Sigma}$. For every part $\hat{p}$ of the value $\hat{v}$ we can obtain a smaller context $\hat{\Sigma}|_{\|\hat{p}\|}$ by erasing all values from the original context $\hat{\Sigma}$ which are not mentioned in $\hat{p}$. The lineage annotations are correct if every part $\hat{p} \sqsubseteq \hat{v}$

$$
\begin{aligned}
[\,M\,]^{\,a}|_b &= \begin{cases} [\,M|_b\,]^{\,a} & \text{if } a \subseteq b \\ [\,] & \text{otherwise} \end{cases} \\
[\,]\,|_b &= [\,] \\
(M +\!\!+ N)|_b &= M|_b +\!\!+ N|_b \\
M^{\cup a}|_b &= \begin{cases} (M|_b)^{\cup a} & \text{if } a \subseteq b \\ [\,] & \text{otherwise} \end{cases} \\
\mathbf{table}\,t|_b &= \mathbf{table}\,t \\
(\mathbf{var}\,x = M; N)|_b &= \mathbf{var}\,x = M|_b; N|_b \\
c|_b &= c \\
(l_i = M_i)^n_{i=1}|_b &= (l_i = M_i|_b)^n_{i=1} \\
M.l|_b &= (M|_b).l \\
(\mathbf{fun}\,f(x_i|^n_{i=1})\,M)|_b &= \mathbf{fun}\,f(x_i|^n_{i=1})\,(M|_b) \\
(\mathbf{if}\,(L)\,M\,\mathbf{else}\,N)|_b &= \mathbf{if}\,(L|_b)\,M|_b\,\mathbf{else}\,N|_b \\
(\mathbf{query}\,M)|_b &= \mathbf{query}\,(M|_b) \\
(\mathbf{for}\,(x <\!- M)\,N)|_b &= \mathbf{for}\,(x <\!- M|_b)\,N|_b \\
(\mathbf{for}\,(x <\!- M)\,N)|_b &= \mathbf{for}\,(x <\!- M|_b)\,N|_b
\end{aligned}
$$

Figure 4.7: Restricting values and terms to those with particular annotations.

of the output $\hat{v}$ is also a part of the output $\hat{v}'$ obtained by evaluating the same query q in the restricted context $\hat{\Sigma}|_{\|\hat{p}\|}$.

Theorem 4.9. *Given monotonic terms M and N, a context $\hat{\Sigma}$, and a set of annotations c, we have*

$$\hat{\Sigma}, M \longrightarrow_{\mathsf{L}} \hat{\Sigma}, N \quad \Rightarrow \quad M|_c = N|_c \quad \vee \quad \hat{\Sigma}|_c, M|_c \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, N|_c$$

Proof. By induction on the evaluation relation $\longrightarrow_{\mathsf{L}}$. We need the alternative $M|_c = N|_c$ because sometimes restriction can yield the empty list, on both sides, in which case there is no evaluation step to be made. The two interesting cases are the singleton **for** comprehension, which introduces $M^{\cup a}$, and adding annotations to a singleton list, which eliminates $M^{\cup a}$.

- Case $\hat{\Sigma}, \mathbf{for}\ (x <\!- [\,V\,]^{\,a})\ M \longrightarrow_{\mathsf{L}} \hat{\Sigma}, M[x := V]^{\cup a}$: We have two cases, depending on c.
  - If $a \subseteq c$ then $(\mathbf{for}\ (x <\!- [\,V\,]^{\,a})\ M)|_c = \mathbf{for}\ (x <\!- [\,V|_c\,]^{\,a})\ (M|_c)$ and therefore $\hat{\Sigma}|_c, \mathbf{for}\ (x <\!- [\,V|_c\,]^{\,a})\ (M|_c) \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, (M|_c[x := V|_c])^{\cup a}$.

$$\frac{}{V \sqsubseteq V} \qquad \frac{}{[\,] \sqsubseteq L} \qquad \frac{V \sqsubseteq V'}{[\,V\,]^{\,b} \sqsubseteq [\,V'\,]^{\,b}} \qquad \frac{V \sqsubseteq V' \qquad W \sqsubseteq W'}{V +\!\!+ W \sqsubseteq V' +\!\!+ W'}$$

$$\frac{\forall 1 \leq i \leq n \qquad l_i = l_i' \qquad V_i \sqsubseteq V_i'}{(l_i = V_i)_{i=1}^{n} \sqsubseteq (l_i' = V_i')_{i=1}^{n}}$$

Figure 4.8: Finding sublists.

Furthermore, we have $(M|_c[x := V|_c])^{\cup a} = ((M[x := V])|_c)^{\cup a}$, which can be shown by induction, but only states that $\cdot|_c$ is well-behaved with respect to substitution, and $((M[x := V])|_c)^{\cup a} = (M[x := V])^{\cup a}|_c$ by definition of $M^{\cup a}|_c$ in the case that $a \subseteq c$, and therefore

$$\hat{\Sigma}|_c, (\mathbf{for}\ (x \mathbin{<\!\!-}\ [\,V\,]^{\,a})\ M)|_c \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, (M[x := V]^{\cup a})|_c$$

– Otherwise $a \not\subseteq c$ and on the left hand side we have

$$\begin{aligned} &\mathbf{for}\ (x \mathbin{<\!\!-}\ [\,V\,]^{\,a})\ M)|_c \\ =\ &\mathbf{for}\ (x \mathbin{<\!\!-}\ ([\,V\,]^{\,a})|_c)\ (M|_c) \\ =\ &\mathbf{for}\ (x \mathbin{<\!\!-}\ [\,]\,)\ (M|_c) \end{aligned}$$

which evaluates to the empty list:

$$\hat{\Sigma}|_c, \mathbf{for}\ (x \mathbin{<\!\!-}\ [\,]\,)\ (M|_c) \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, [\,]$$

Since $(M[x := V]^{\cup a})|_c = [\,]$ we can conclude that

$$\hat{\Sigma}|_c, (\mathbf{for}\ (x \mathbin{<\!\!-}\ [\,V\,]^{\,a})\ M)|_c \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, (M[x := V]^{\cup a})|_c$$

- Case $\hat{\Sigma}, ([\,V\,]^{\,b})^{\cup a} \longrightarrow_{\mathsf{L}} \hat{\Sigma}, [\,V\,]^{\,a \cup b}$: We have two cases depending on c.

  – If $a \subseteq c$ then $([\,V\,]^{\,b})^{\cup a}|_c = ([\,V\,]^{\,b}|_c)^{\cup a}$. Now, if $b \subseteq c$ then $[\,V\,]^{\,b}|_c = [\,V|_c\,]^{\,b}$ and we have an evaluation step $\hat{\Sigma}|_c, ([\,V|_c\,]^{\,b})^{\cup a} \longrightarrow_{\mathsf{L}} \hat{\Sigma}|_c, [\,V|_c\,]^{\,a \cup b}$ where the term on the right hand side is equal to $[\,V\,]^{\,a \cup b}|_c$.

  – Otherwise, $b \not\subseteq c$ and $[\,V\,]^{\,b}|_c = [\,]$ but on the right hand side we also have $[\,V\,]^{\,a \cup b}|_c = [\,]$. In other words, by restricting with c we get the same value on both sides. We reach the same conclusion in the case that $a \not\subseteq c$. □

Corollary 4.10. *By repeated application of Theorem 4.9 we have*

$$\hat{\Sigma}, M \longrightarrow_{\mathsf{L}}^{j} \hat{\Sigma}, N \quad \Rightarrow \quad \hat{\Sigma}|_c, M|_c \longrightarrow_{\mathsf{L}}^{k} \hat{\Sigma}|_c, N|_c$$

where $j, k \in \mathbb{N}$ *and* $k \leq j$.

Lemma 4.11. *Given a value* $\hat{v}$ *and a subvalue* $\hat{p} \sqsubseteq \hat{v}$ *of that value, we have*

$$\hat{p} \sqsubseteq \hat{v}|_{\|\hat{p}\|}$$

Proof. By induction on the subvalue relation $\sqsubseteq$.

- Cases $V \sqsubseteq V$ and $[\,] \sqsubseteq V$ are trivially true.
- Case $[V]^b \sqsubseteq [V']^b$: We have $[V']^b|_{\|[V]^b\|} = [V']^b|_{b \cup \|V\|}$ by definition, and $V'|_{\|V\|} \sqsupseteq V$ by the induction hypothesis, and can therefore conclude $[V]^b \sqsubseteq [V']^b|_{\|[V]^b\|}$.
- The cases for list concatenation and records are similar. □

Theorem 4.12 (Correctness of lineage)**.** *Let* q *be a monotonic query with* $\|q\| = \emptyset$ *and let* $\hat{\Sigma}$ *be a context, such that* q *evaluates to* $\hat{v}$ *in* $\hat{\Sigma}$*:* $\hat{\Sigma}, q \longrightarrow_{\mathsf{L}}^{*} \hat{\Sigma}, \hat{v}$*. Then for every sublist* $\hat{p} \sqsubseteq \hat{v}$ *we can evaluate* q *in a restricted context* $\hat{\Sigma}|_{\|\hat{p}\|}$ *to obtain a value* $\hat{v}'$ *and* $\hat{p}$ *will be a sublist of* $\hat{v}'$*.*

$$\forall \hat{p} \sqsubseteq \hat{v} : \quad \hat{\Sigma}|_{\|\hat{p}\|}, q \longrightarrow_{\mathsf{L}}^{*} \hat{\Sigma}|_{\|\hat{p}\|}, \hat{v}' \quad \wedge \quad \hat{p} \sqsubseteq \hat{v}'$$

Proof. Using Corollary 4.10 of Theorem 4.9 we have $\hat{\Sigma}|_{\|\hat{p}\|}, q|_{\|\hat{p}\|} \longrightarrow_{\mathsf{L}}^{*} \hat{\Sigma}|_{\|\hat{p}\|}, \hat{v}|_{\|\hat{p}\|}$ for any $\hat{p}$ and, because of Lemma 4.11, $\hat{v}|_{\|\hat{p}\|} \sqsupseteq \hat{p}$ so set $\hat{v}' = \hat{v}|_{\|\hat{p}\|}$. Since q has no annotations on its own, it is not affected by restriction: $q|_{\|\hat{p}\|} = q$ and we can conclude that $\hat{\Sigma}|_{\|\hat{p}\|}, q \longrightarrow_{\mathsf{L}}^{*} \hat{\Sigma}|_{\|\hat{p}\|}, \hat{v}' \wedge \hat{p} \sqsubseteq \hat{v}'$. □

This concludes the syntax and semantics of Links$^{\mathsf{L}}$. Tables implicitly carry initial lineage annotations on their rows. Lineage annotations are propagated and combined inside **`lineage`** blocks, modeled by the extended evaluation relation $\longrightarrow_{\mathsf{L}}$, but the result of a lineage query is reified into a plain Links value. Finally, we consider lineage correct if we can run a query, remove all values from the database that do not appear as lineage annotations in the query result, and get the same result. Next, we discuss one implementation strategy for Links$^{\mathsf{L}}$, a translation from Links$^{\mathsf{L}}$ to plain Links.

$$
\begin{array}{rcl}
\mathfrak{D}[\![O]\!] & = & O \\
\mathfrak{D}[\![A \text{ -> } B]\!] & = & (\mathfrak{D}[\![A]\!] \text{ -> } \mathfrak{D}[\![B]\!], \mathfrak{L}[\![A]\!] \text{ -> } \mathfrak{L}[\![B]\!]) \\
\mathfrak{D}[\![(l_i : A_i)^{\,n}_{i=1}]\!] & = & (l_i : \mathfrak{D}[\![A_i]\!])^{\,n}_{i=1} \\
\mathfrak{D}[\![[A]]\!] & = & [\mathfrak{D}[\![A]\!]] \\
\mathfrak{D}[\![\texttt{table}(R)]\!] & = & (\texttt{table}(R), () \text{ -> } \mathfrak{L}[\![(R)]\!])
\end{array}
$$

Figure 4.13: Doubling translation on types.

4.3 Implementation

We define a translation from Links^{L} to Links. The translation has two parts: an outer translation called the *doubling translation* ($\mathfrak{D}$) and an inner translation called the *lineage translation* ($\mathfrak{L}$). The former is used for translating ordinary Links^{L} code while the latter is used to translate query code inside a **`lineage`** block. In addition to doubling up tables with a view that generates initial annotations, we also double functions: one version for unannotated evaluation and one version that propagates lineage annotations. Since functions and queries can close over variables, we use the $\mathfrak{L}^*$ variant of the lineage translation to select the lineage variant of doubled functions and initialize external lists with empty lineage.

The syntactic translation of Links^{L} types for the doubling translation is shown in Figure 4.13. For the lineage translation, we use the same $\mathfrak{L}$ translation shown earlier in Figure 4.2. We write $\mathfrak{D}[\![\Gamma]\!]$ and $\mathfrak{L}[\![\Gamma]\!]$ for the obvious extensions of these translations to contexts.

The translation of Links^{L} expressions to Links is shown in Figures 4.14–4.16. Following the type translation, term translation operates in two modes: $\mathfrak{D}$ and $\mathfrak{L}$. We translate the syntax of ordinary Links programs using the translation $\mathfrak{D}[\![-]\!]$. When we reach a **`lineage`** block, we switch to using the $\mathfrak{L}[\![-]\!]$ translation. $\mathfrak{L}[\![[M]]\!]$ provides initial lineage for list literals. Their lineage is simply empty. Table comprehension is the most interesting case. We translate a table iteration **`for`** $(x \texttt{ <-- } L)$ M to a nested list comprehension. The outer comprehension binds y to the results of the lineage-computing view of L. The inner comprehension binds a fresh variable z, iterating over $\mathfrak{L}[\![M]\!]$—the original comprehension body M transformed using $\mathfrak{L}$. The original comprehension body M is defined in terms of x, which is not bound in the transformed comprehension. We therefore

$$
\begin{array}{rcl}
\mathfrak{D}[\![c]\!] & = & c \\
\mathfrak{D}[\![x]\!] & = & x \\
\mathfrak{D}[\![(l_i = M_i)_{i=1}^n]\!] & = & (l_i = \mathfrak{D}[\![M_i]\!])_{i=1}^n \\
\mathfrak{D}[\![N.l]\!] & = & \mathfrak{D}[\![N]\!].l \\
\mathfrak{D}[\![\mathtt{fun}(x_i|_{i=1}^n)\ \{M\}]\!] & = & (\mathtt{fun}(x_i|_{i=1}^n)\ \{\mathfrak{D}[\![M]\!]\}, \mathfrak{L}^*[\![\mathtt{fun}(x_i|_{i=1}^n)\ \{M\}]\!]) \\
\mathfrak{D}[\![M(N_i|_{i=1}^n)]\!] & = & \mathfrak{D}[\![M]\!].1(\mathfrak{D}[\![N_i]\!]_{i=1}^n) \\
\mathfrak{D}[\![\mathtt{var}\ x = M; N]\!] & = & \mathtt{var}\ x = \mathfrak{D}[\![M]\!]; \mathfrak{D}[\![N]\!] \\
\mathfrak{D}[\![\,[\,]\,]\!] & = & [\,] \\
\mathfrak{D}[\![\,[M]\,]\!] & = & [\mathfrak{D}[\![M]\!]] \\
\mathfrak{D}[\![M \mathbin{++} N]\!] & = & \mathfrak{D}[\![M]\!] \mathbin{++} \mathfrak{D}[\![N]\!] \\
\mathfrak{D}[\![\mathtt{if}\ (L)\ \{M\}\ \mathtt{else}\ \{N\}]\!] & = & \mathtt{if}\ (\mathfrak{D}[\![L]\!])\ \{\mathfrak{D}[\![M]\!]\}\ \mathtt{else}\ \{\mathfrak{D}[\![N]\!]\} \\
\mathfrak{D}[\![\mathtt{query}\ \{M\}]\!] & = & \mathtt{query}\ \{\mathfrak{D}[\![M]\!]\} \\
\mathfrak{D}[\![\mathtt{empty}\ (M)]\!] & = & \mathtt{empty}\ (\mathfrak{D}[\![M]\!]) \\
\mathfrak{D}[\![\mathtt{for}\ (x <\!- L)\ M]\!] & = & \mathtt{for}\ (x <\!- \mathfrak{D}[\![L]\!])\ \mathfrak{D}[\![M]\!] \\
\mathfrak{D}[\![\mathtt{where}(M)\ N]\!] & = & \mathtt{where}(\mathfrak{D}[\![M]\!])\ \mathfrak{D}[\![N]\!] \\
\mathfrak{D}[\![\mathtt{for}\ (x <\!-\!- L)\ M]\!] & = & \mathtt{for}\ (x <\!- \mathfrak{D}[\![L]\!].1)\ \mathfrak{D}[\![M]\!] \\
\mathfrak{D}[\![\mathtt{insert}\ L\ \mathtt{values}\ M]\!] & = & \mathtt{insert}\ \mathfrak{D}[\![L]\!].1\ \mathtt{values}\ \mathfrak{D}[\![M]\!] \\
\mathfrak{D}[\![\mathtt{update}\ (x <\!- L)\ \mathtt{where}\ M\ \mathtt{set}\ \mathfrak{D}[\![N]\!] & = & \mathtt{update}\ (x <\!- \mathfrak{D}[\![L]\!].1)\ \mathtt{where}\ \mathfrak{D}[\![M]\!]\ \mathtt{set}\ N]\!] \\
\mathfrak{D}[\![\mathtt{delete}\ (x <\!- L)\ \mathtt{where}\ M]\!] & = & \mathtt{delete}\ (x <\!- \mathfrak{D}[\![L]\!].1)\ \mathtt{where}\ \mathfrak{D}[\![M]\!] \\
\mathfrak{D}[\![\mathtt{lineage}\ \{M\}]\!] & = & \mathtt{query}\ \{\mathfrak{L}^*[\![M]\!]\} \\
\mathfrak{D}[\![\mathtt{table}\ n\ \mathtt{with}\ (R)]\!] & = & (\mathtt{table}\ n\ \mathtt{with}\ (R),\ \ \mathtt{fun}()\{\mathfrak{L}[\![\mathtt{table}\ n\ \mathtt{with}\ (R)]\!]\})
\end{array}
$$

Figure 4.14: Translation of Links$^{\mathrm{L}}$ to Links: outer translation.

replace every occurrence of x in $\mathfrak{L}[\![M]\!]$ by y`.data`. In the body of the nested comprehension we thus have y, referring to the table row annotated with lineage, and z, referring to the result of the original comprehension's body, also annotated with lineage. As the result of our transformed comprehension, we return the plain data part of z as our data, and the combined lineage annotations of y and z as our provenance.

One subtlety here is that lineage blocks need not be closed, and so may refer to variables that were defined (and will be bound to values at runtime) outside of the lineage block. This could cause problems: for example, if we bind x to a collection $[1,2,3]$ outside a lineage block and refer to it in a comprehension inside such a block, then uses of x will expect the collection elements to be

$$
\begin{array}{rcl}
\mathfrak{L}[\![c]\!] & = & c \\
\mathfrak{L}[\![x]\!] & = & x \\
\mathfrak{L}[\![(l_i = M_i)_{i=1}^n]\!] & = & (l_i = \mathfrak{L}[\![M_i]\!])_{i=1}^n \\
\mathfrak{L}[\![N.l]\!] & = & \mathfrak{L}[\![N]\!].l \\
\mathfrak{L}[\![\mathbf{fun}(x_i|_{i=1}^n)\ \{M\}]\!] & = & (\mathbf{fun}(x_i|_{i=1}^n)\ \{\mathfrak{L}[\![M]\!]\}) \\
\mathfrak{L}[\![M(N_i|_{i=1}^n)]\!] & = & \mathfrak{L}[\![M]\!](\mathfrak{L}[\![N_i]\!]|_{i=1}^n) \\
\mathfrak{L}[\![\mathbf{var}\ x = M; N]\!] & = & \mathbf{var}\ x = \mathfrak{L}[\![M]\!]; \mathfrak{L}[\![N]\!] \\
\mathfrak{L}[\![\texttt{[]}]\!] & = & \texttt{[]} \\
\mathfrak{L}[\![\texttt{[}M\texttt{]}]\!] & = & \texttt{[(data =}\ \mathfrak{L}[\![M]\!], \mathsf{prov} = \texttt{[])]} \\
\mathfrak{L}[\![M \mathbin{++} N]\!] & = & \mathfrak{L}[\![M]\!] \mathbin{++} \mathfrak{L}[\![N]\!] \\
\mathfrak{L}[\![\mathbf{if}\ (L)\ \{M\}\ \mathbf{else}\ \{N\}]\!] & = & \mathbf{if}\ (\mathfrak{L}[\![L]\!])\ \{\mathfrak{L}[\![M]\!]\}\ \mathbf{else}\ \{\mathfrak{L}[\![N]\!]\} \\
\mathfrak{L}[\![\mathbf{query}\ \{M\}]\!] & = & \mathbf{query}\ \{\mathfrak{L}[\![M]\!]\} \\
\mathfrak{L}[\![\mathbf{empty}\ (M)]\!] & = & \mathbf{empty}\ (\mathfrak{L}[\![M]\!]) \\
\mathfrak{L}[\![\mathbf{for}\ (x <\!\!- L)\ M]\!] & = & \mathbf{for}\ (y <\!\!- \mathfrak{L}[\![L]\!]) \\
 & & \quad \mathbf{for}\ (z <\!\!- \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}]) \\
 & & \quad\quad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \mathbin{++} z.\mathsf{prov})] \\
\mathfrak{L}[\![\mathbf{where}(M)\ N]\!] & = & \mathbf{where}(\mathfrak{L}[\![M]\!])\ (\mathfrak{L}[\![N]\!]) \\
\mathfrak{L}[\![\mathbf{for}\ (x <\!\!-\!\!- L)\ M]\!] & = & \mathbf{for}\ (y <\!\!- \mathfrak{L}[\![L]\!]) \\
 & & \quad \mathbf{for}\ (z <\!\!- \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}]) \\
 & & \quad\quad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \mathbin{++} z.\mathsf{prov})] \\
\mathfrak{L}[\![\mathbf{lineage}\ \{M\}]\!] & = & \mathbf{query}\ \{\mathfrak{L}[\![M]\!]\} \\
\mathfrak{L}[\![\mathbf{table}\ n\ \mathbf{with}\ (R)]\!] & = & \mathbf{for}(x <\!\!-\!\!- \mathbf{table}\ n\ \mathbf{with}\ (R))[(\mathsf{data} = x, \mathsf{prov} = [(n, x.\mathsf{oid})])]
\end{array}
$$

Figure 4.15: Translation of LINKS$^{\mathrm{L}}$ to LINKS: inner translation.

records such as $(\mathsf{data} = 1, \mathsf{prov} = L)$ rather than plain numbers. Therefore, such variables need to be adjusted so that they will have appropriate structure to be used within a lineage block. The auxiliary type-indexed function $d2l[\![A]\!]$ in Figure 4.16 accomplishes this by mapping a value of type $\mathfrak{D}[\![A]\!]$ to one of type $\mathfrak{L}[\![A]\!]$. We define $\mathfrak{L}^*[\![-]\!]$ as a function that applies $\mathfrak{L}[\![-]\!]$ to its argument and substitutes all free variables $x : A$ with $d2l[\![A]\!](x)$.

The $\mathfrak{D}[\![-]\!]$ translation also has to account for functions that are defined outside lineage blocks but may be called either outside or inside a lineage block. To support this, the case for functions in the $\mathfrak{D}[\![-]\!]$ translation creates a pair, whose first component is the recursive $\mathfrak{D}[\![-]\!]$ translation of the function, and

$$
\begin{array}{rcl}
\mathfrak{L}^*[\![M]\!] & = & \mathfrak{L}[\![M]\!][x_i \mapsto d2l[\![A_i]\!](x_i)|_{i=1}^n] \\
 & & \text{where } x_1 : A_1, \ldots, x_n : A_n \text{ are the free variables of } M \\
d2l[\![A]\!] & : & \mathfrak{D}[\![A]\!] \text{ -> } \mathfrak{L}[\![A]\!] \\
d2l[\![O]\!](x) & = & x \\
d2l[\![A \text{ -> } B]\!](f) & = & f.2 \\
d2l[\![(l_1 : A_1, \ldots, l_n : A_n)]\!](x) & = & (l_1 : d2l[\![A_1]\!](x.l_1), \ldots, l_n : d2l[\![A_n]\!](x.l_n)) \\
d2l[\![[A]]\!](y) & = & \mathbf{for}(x \leftarrow y)[(\mathsf{data} = d2l[\![A]\!](x), \mathsf{prov} = [])] \\
d2l[\![\mathbf{table}(R)]\!](t) & = & t.2()
\end{array}
$$

Figure 4.16: Translation of Links$^{\mathrm{L}}$ to Links: term translation.

whose second component uses the $\mathfrak{L}^*[\![-]\!]$ translation to create a version of the function callable from within a lineage block. (We use $\mathfrak{L}^*[\![-]\!]$ because functions also need not be closed.) Function calls outside lineage blocks are translated to project out the first component; function calls inside such blocks are translated to project out the second component (this is actually accomplished via the A -> B case of $d2l$.) Finally, the $\mathfrak{D}[\![-]\!]$ translation maps table types and table references to pairs. This is similar to the $\mathfrak{W}[\![-]\!]$ translation, so we do not explain it in further detail; the main difference is that we just use the `oid` column to assign default provenance to all rows.

For example, if we wrap the query from Figure 2.4 on page 27 in a **`lineage`** block it will be rewritten to this:

```
for (y_e <- externalTours.2())
for (z_e <- where (y_e.data.type == "boat")
              for (y_a <- agencies.2())
              for (z_a <- where (y_a.data.name == y_e.data.name)
                                [(data = (name = y_e.data.name,
                                          phone = y_a.data.phone),
                                  prov = [])])
                [(data = z_a.data, prov = y_a.prov ++ z_a.prov)])
  [(data = z_e.data, prov = y_e.prov ++ z_e.prov)]
```

Once `agencies` and `externalTours` are inlined, Links's built-in normalization algorithm simplifies this query to:

```
for (y_a <- table "Agencies" with ...)
  for (y_e <- table "ExternalTours" with ...)
  where (y_a.data.name == y_e.data.name && y_e.data.type == "boat")
```

```
[(data = (name = y_a.data.name,phone = y_a.data.phone),
  prov = [("Agencies", y_a.oid), ("ExternalTours",y_e.oid)])]
```

We see that normalization did its job, partially evaluating the initial lineage annotation views, removing unnecessary nesting of comprehensions, and accumulating conditions in a single where clause. From there, query shredding [Cheney et al., 2014c] will generate two reasonably efficient SQL queries.

Before considering the main result, we state an auxiliary lemma:

Lemma 4.17. *If* $\Gamma \vdash M : \mathfrak{D}[\![A]\!]$ *then* $\Gamma \vdash d2l(M) : \mathfrak{L}[\![A]\!]$.

Proof. The proof is by induction on the structure of A but each case is straightforward. We show the interesting cases for function types and collection types:

- If $A = B_1 \texttt{->} B_2$ then we proceed as follows:

$$\frac{\Gamma \vdash M : (\mathfrak{D}[\![B_1]\!] \texttt{->} \mathfrak{D}[\![B_2]\!], \mathfrak{L}[\![B_1]\!] \texttt{->} \mathfrak{L}[\![B_2]\!])}{\Gamma \vdash M.2 : \mathfrak{L}[\![B_1]\!] \texttt{->} \mathfrak{L}[\![B_2]\!]}$$

  which suffices since $\mathfrak{L}[\![B_1 \texttt{->} B_2]\!] = \mathfrak{L}[\![B_1]\!] \texttt{->} \mathfrak{L}[\![B_2]\!]$.

- If $A = \texttt{[}B\texttt{]}$ then we proceed as follows:

$\Gamma \vdash M : \texttt{[}\mathfrak{D}[\![B]\!]\texttt{]}$	assumption
$\Gamma' \vdash x : \mathfrak{D}[\![B]\!]$	rule
$\Gamma' \vdash d2l[\![B]\!](x) : \mathfrak{L}[\![B]\!]$	IH
$\Gamma' \vdash \texttt{[]} : \texttt{[(String,Int)]}$	rule
$\Gamma' \vdash \texttt{(data} = d2l[\![B]\!](x), \texttt{prov} = \texttt{[])} : \textbf{Lin}(\mathfrak{L}[\![B]\!])$	rule
$\Gamma' \vdash \texttt{[(data} = d2l[\![B]\!](x), \texttt{prov} = \texttt{[])]} : \texttt{[}\textbf{Lin}(\mathfrak{L}[\![B]\!])\texttt{]}$	rule
$\Gamma \vdash \textbf{for}\ (x \texttt{<-} M)\ \texttt{[(data} = d2l[\![B]\!](x), \texttt{prov} = \texttt{[])]} : \texttt{[}\textbf{Lin}(\mathfrak{L}[\![B]\!])\texttt{]}$	rule

  where $\Gamma' = \Gamma, x : \mathfrak{D}[\![B]\!]$. □

The type-preservation property for the translation from LINKS^{L} to LINKS is stated as follows:

Theorem 4.18. *Let* M *be given such that* $\Gamma \vdash_{\text{LINKS}^{L}} M : A$. *Then:*

1. $\mathfrak{L}[\![\Gamma]\!] \vdash_{\text{LINKS}} \mathfrak{L}[\![M]\!] : \mathfrak{L}[\![A]\!]$
2. $\mathfrak{D}[\![\Gamma]\!] \vdash_{\text{LINKS}} \mathfrak{L}^{*}[\![M]\!] : \mathfrak{L}[\![A]\!]$
3. $\mathfrak{D}[\![\Gamma]\!] \vdash_{\text{LINKS}} \mathfrak{D}[\![M]\!] : \mathfrak{D}[\![A]\!]$

Proof. For the first part, the proof is by induction on the structure of the typing derivation. The interesting cases are the List, ForList, and ForTable cases, where lineage annotations are created or combined. We show the details of the cases for singleton lists and list comprehensions. Table comprehensions are similar.

- If the derivation is of the form:

$$\textsc{List}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{[}M\texttt{]} : \texttt{[}A\texttt{]}}$$

then we proceed as follows:

$\mathfrak{L}[\![\Gamma]\!] \vdash \mathfrak{L}[\![M]\!] : \mathfrak{L}[\![A]\!]$	by IH
$\mathfrak{L}[\![\Gamma]\!] \vdash \texttt{[]} : \texttt{[(String,Int)]}$	by rule
$\mathfrak{L}[\![\Gamma]\!] \vdash \texttt{(data} = \mathfrak{L}[\![M]\!], \texttt{prov} = \texttt{[])} : \mathbf{Lin}(\mathfrak{L}[\![A]\!])$	by rule
$\mathfrak{L}[\![\Gamma]\!] \vdash \texttt{[(data} = \mathfrak{L}[\![M]\!], \texttt{prov} = \texttt{[])]} : \texttt{[}\mathbf{Lin}(\mathfrak{L}[\![A]\!])\texttt{]}$	by rule

which suffices since

$$\mathfrak{L}[\![\,\texttt{[}A\texttt{]}\,]\!] = \texttt{[}\mathbf{Lin}\mathfrak{L}[\![A]\!]\texttt{]} = \texttt{[(data} : \mathfrak{L}[\![A]\!], \texttt{prov} : \texttt{[(String,Int)])]}$$

- If the derivation is of the form:

$$\textsc{For-List}\ \frac{\Gamma \vdash L : \texttt{[}A\texttt{]} \qquad \Gamma, x : A \vdash M : \texttt{[}B\texttt{]}}{\Gamma \vdash \mathbf{for}\ (x \texttt{<-}\ L)\ M : \texttt{[}B\texttt{]}}$$

then we proceed as follows,writing Γ' for $\mathfrak{L}[\![\Gamma]\!], y : \mathbf{Lin}(\mathfrak{L}[\![A]\!]), z : \mathbf{Lin}(\mathfrak{L}[\![B]\!])$:

$$
\begin{array}{ll}
\mathfrak{L}[\![\Gamma]\!] \vdash \mathfrak{L}[\![L]\!] : [\,\mathbf{Lin}(\mathfrak{L}[\![A]\!])\,] & \text{IH} \\
\mathfrak{L}[\![\Gamma]\!], x : \mathfrak{L}[\![A]\!] \vdash \mathfrak{L}[\![M]\!] : [\,\mathbf{Lin}(\mathfrak{L}[\![B]\!])\,] & \text{IH} \\
\mathfrak{L}[\![\Gamma]\!], y : \mathbf{Lin}(\mathfrak{L}[\![A]\!]) \vdash y.\mathtt{data} : \mathfrak{L}[\![A]\!] & \text{rule} \\
\mathfrak{L}[\![\Gamma]\!], y : \mathbf{Lin}(\mathfrak{L}[\![A]\!]) \vdash \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}] : \mathbf{Lin}(\mathfrak{L}[\![A]\!]) & \text{subst} \\
\Gamma' \vdash z.\mathsf{data} : \mathfrak{L}[\![B]\!] & \text{rule} \\
\Gamma' \vdash y.\mathsf{prov} : [\,(\mathtt{String}, \mathtt{Int})\,] & \text{rule} \\
\Gamma' \vdash z.\mathsf{prov} : [\,(\mathtt{String}, \mathtt{Int})\,] & \text{rule} \\
\Gamma' \vdash y.\mathsf{prov} \mathbin{+\!\!+} z.\mathsf{prov} : [\,(\mathtt{String}, \mathtt{Int})\,] & \text{rule} \\
\Gamma' \vdash (\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \mathbin{+\!\!+} z.\mathsf{prov}) : \mathbf{Lin}(\mathfrak{L}[\![B]\!]) & \text{rule} \\
\Gamma' \vdash [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \mathbin{+\!\!+} z.\mathsf{prov})] : [\,\mathbf{Lin}(\mathfrak{L}[\![B]\!])\,] & \text{rule} \\
\mathfrak{L}[\![\Gamma]\!], y : \mathbf{Lin}(\mathfrak{L}[\![A]\!]) \vdash & \\
\quad \mathbf{for}\ (z <\!\!- \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}]) & \text{rule} \\
\quad\quad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \mathbin{+\!\!+} z.\mathsf{prov})] : [\,\mathbf{Lin}(\mathfrak{L}[\![B]\!])\,] & \\
\mathfrak{L}[\![\Gamma]\!] \vdash\ \mathbf{for}\ (y <\!\!- \mathfrak{L}[\![L]\!]) & \text{rule} \\
\quad \mathbf{for}\ (z <\!\!- \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}]) & \\
\quad\quad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \mathbin{+\!\!+} z.\mathsf{prov})] : [\,\mathbf{Lin}(\mathfrak{L}[\![B]\!])\,] &
\end{array}
$$

For the second part, suppose $\Gamma \vdash M : A$. Then by part 1 we know $\mathfrak{L}[\![\Gamma]\!] \vdash \mathfrak{L}[\![M]\!] : \mathfrak{L}[\![A]\!]$. Clearly, for each $x_i : A_i$ in Γ we have $\mathfrak{D}[\![\Gamma]\!] \vdash x_i : \mathfrak{D}[\![A_i]\!]$, so it follows that $\mathfrak{D}[\![\Gamma]\!] \vdash d2l(x_i) : \mathfrak{L}[\![A_i]\!]$ for each i by Lemma 4.17. Using the (standard) substitution lemma for LINKS typing, we can conclude $\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{L}^*[\![M]\!] : \mathfrak{L}[\![A]\!]$.

Finally, for the third part, again the proof is by induction on the structure of the derivation of $\Gamma \vdash M : A$. Most cases are straightforward; we show the interesting cases for functions, function application, and **lineage**, illustrating the need for duplicating code in the type translation for functions and the use of $\mathfrak{L}^*[\![-]\!]$. The cases for updates and table references are similar to those for LINKS$^{\mathrm{W}}$, but simpler because the types of the fields do not change in the translation from LINKS$^{\mathrm{L}}$ to LINKS.

- If the derivation is of the form

$$\dfrac{\textsc{Fun}\quad \Gamma, x : A \vdash M : B}{\Gamma \vdash \mathbf{fun}\ (x)\ \{M\} : A \mathrel{-\!\!>} B}$$

  then by induction we have $\mathfrak{D}[\![\Gamma]\!], x : \mathfrak{D}[\![A]\!] \vdash \mathfrak{D}[\![M]\!] : \mathfrak{D}[\![B]\!]$ and by part 2 we know that $\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{L}^*[\![\mathbf{fun}\ (x)\ \{M\}]\!] : \mathfrak{L}[\![\ (A)\ \mathrel{-\!\!>} B]\!]$. We can proceed as

follows:

$$\begin{array}{ll}
\mathfrak{D}[\![\Gamma]\!], x : \mathfrak{D}[\![A]\!] \vdash \mathfrak{D}[\![M]\!] : \mathfrak{D}[\![B]\!] & \text{by IH} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \mathbf{fun}\ (x)\ \{\mathfrak{D}[\![M]\!]\} : \mathfrak{D}[\![A]\!] \text{ -> } \mathfrak{D}[\![B]\!] & \text{by rule} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{L}^*[\![\mathbf{fun}\ (x)\ \{M\}]\!] : \mathfrak{L}[\![A]\!] \text{ -> } \mathfrak{L}[\![B]\!] & \text{by part 2} \\
\mathfrak{D}[\![\Gamma]\!] \vdash (\mathbf{fun}\ (x)\ \{\mathfrak{D}[\![M]\!]\}, \mathfrak{L}^*[\![\mathbf{fun}\ (x)\ \{M\}]\!]) : \mathfrak{D}[\![A \text{ -> } B]\!] & \text{by rule}
\end{array}$$

where the final step relies on the fact that

$$\mathfrak{D}[\![A \text{ -> } B]\!] = (\mathfrak{D}[\![A]\!] \text{ -> } \mathfrak{D}[\![B]\!], \mathfrak{L}[\![A]\!] \text{ -> } \mathfrak{L}[\![B]\!])$$

- If the derivation is of the form

$$\textsc{App}\ \frac{\Gamma \vdash M : A \text{ -> } B \qquad \Gamma \vdash N : A}{\Gamma \vdash M(N) : B}$$

then we proceed as follows:

$$\begin{array}{ll}
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{D}[\![M]\!] : (\mathfrak{D}[\![A]\!] \text{ -> } \mathfrak{D}[\![B]\!], \mathfrak{L}[\![A]\!] \text{ -> } \mathfrak{L}[\![B]\!]) & \text{by IH} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{D}[\![M]\!].1 : \mathfrak{D}[\![A]\!] \text{ -> } \mathfrak{D}[\![B]\!] & \text{by rule} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{D}[\![N]\!] : \mathfrak{D}[\![A]\!] & \text{by IH} \\
\mathfrak{D}[\![\Gamma]\!] \vdash \mathfrak{D}[\![M]\!].1(\mathfrak{D}[\![N]\!]) : \mathfrak{D}[\![B]\!] & \text{by rule}
\end{array}$$

where in the first step we use the fact that

$$\mathfrak{D}[\![A \text{ -> } B]\!] = (\mathfrak{D}[\![A]\!] \text{ -> } \mathfrak{D}[\![B]\!], \mathfrak{L}[\![A]\!] \text{ -> } \mathfrak{L}[\![B]\!])$$

- Nested lineage blocks are currently not supported as annotating lineage annotations with lineage annotations would just duplicate the same annotations. □

As with the where-provenance translation, we have proven the correctness of lineage annotation propagation (Theorem 4.12) and type-preservation of the translation (Theorem 4.18). The latter is a partial sanity check, but no proof, that this translation faithfully implements the semantics. A simulation proof in the style of Theorem 3.10 should intuitively work similarly. However, the **`lineage`** block relies on the lineage annotation propagation operator $M^{\cup b}$ in the Links$^{\mathrm{W}}$ semantics, which does not have a direct translation to plain Links.

4.4 Discussion

Links^{L} is inspired by prior work on lineage by Cui et al. [2000] and why-provenance by Buneman et al. [2001]. Almost all of the provenance systems mentioned in Section 2.1.2 implement either lineage or some form of why-provenance.

DBNotes [Chiticariu et al., 2005] does not natively implement lineage but we believe it can be encoded using its support for custom annotation propagation. DBNotes in particular shows some interesting parallels to Links^{L}. DBNotes may generate multiple Sql queries from a single query with annotation propagation. Links^{L} ultimately also generates as many queries as there are list constructors in the result type. DBNotes uses a post-processing step to construct nested collections of annotations from flat query results to display to the user. Links^{L} also internally stitches flat query results together into a nested result. The difference is that Links^{L} gets all of this for free by piggybacking on Links's support for nested multisets via query shredding [Cheney et al., 2014c].

This native support for nested multisets leads to the very natural — if not plain obvious — encoding of lineage as multisets of annotations on every result row that we use in Links^{L}. Lineage fits so neatly into the nested multiset data model that further processing of annotations is just the same as working with regular data. We can thus claim to fulfill Glavic et al.'s Requirement 3.

We compare the performance of Links^{L} to Perm in Section 5.3.2. Lee et al. [2018] compared the performance of Links^{L} to Pug using some of the same queries. We discuss their results briefly in Section 5.4. In terms of Requirement 4, we can say that lineage computation scales to large databases since it grows proportionally to the query result, not the whole database and is computed on demand for those parts of the database that are relevant to the result.

In terms of type-safety, Links^{L} does not go as far as Links^{W} with its abstract type for where-provenance-annotated data. It is possible to accidentally have fewer annotations than intended. Consider the following code snippet.

```
var xs = query { for (x <-- table "xs" ...) f(x) };
lineage { for (x <- xs) g(x) }
```

Because `xs` is defined outside of the **`lineage`** block, it will be assigned empty annotations when it is used in the lineage query. Thus the result will not have annotations referring to the *"xs"* table (unless added by `g`). Plain Links already has essentially the same problem — such queries have suboptimal performance

— with a easy solutions: enclose the whole expression in a **`query`** block or wrap it in a thunk. One could even argue that this is a feature: it allows users to not compute lineage for the parts of a query they trust.

In exchange for its weaker correctness guarantees Links^{L} offers increased convenience. Changing **`query`** to **`lineage`** is all that is necessary to propagate lineage through a whole query. Links^{L} automatically adds annotations to source tables and propagates them even through user-defined functions. To make this work we create a second version of every function which handles lineage. This is similar to the doubling translation used by Cheney et al. [2014b] to compile a simplified form of Links to a F#-like core language. Both translations introduce space overhead and overhead for normal function calls due to pair projections. Developing a more efficient alternative translation is an interesting topic for future work, perhaps in combination with a strategy for compiling Links to native code. The difficulty here is that queries can be constructed at runtime using higher-order functions. We should also note that the current prototype implementation of Links^{L} does not handle doubling of functions correctly. To run the experiments in the next chapter we manually inlined all functions.

The prototype implementation of Links^{L} uses the automatically generated `oid` column to identify rows for simplicity. This is easily generalized to custom annotations per table, like in Links^{W}, and has been implemented by Stolarek and Cheney [2018]. In other provenance systems, for example Pug [Lee et al., 2018], lineage contains not only a pointer to the original data, but the data itself. Since lineage annotations can come from multiple source tables with different types, doing the same is difficult in a well-typed system. This is not a problem in Pug where lineage annotations are strings, but we do not consider strings a satisfying solution for Links^{L}. If Links supported variants in query results, we could possibly use those for lineage annotations of varying types.

Chapter 5

Performance evaluation

This chapter includes material from previously published work [Fehrenbach and Cheney, 2016, 2018].

This chapter considers the performance of language-integrated provenance and in particular that of our prototype implementations of Links^{W} and Links^{L}. In Sections 5.1 and 5.2 we compare them against plain Links to determine the overhead of tracing where-provenance and lineage, respectively. In Section 5.3 we compare them against Perm, a database-integrated provenance system. Section 5.4 discusses threats to validity, performance-related aspects of the prototype implementations and related work, and benchmarks performed by [Lee et al., 2018] comparing Links^{L} to Pug.

Support for nested relational queries is one of the distinguishing features of language-integrated provenance compared to other provenance systems and particularly important in Links^{L}. We are not aware of a commonly accepted standard benchmark for nested relational systems, so we turn to the work on query shredding as a source of benchmark data and nested queries. Cheney et al. [2014c] use queries against a simple test database schema (see Figure 5.1) that models an organization with departments, employees and external contacts.[1] "Each department has a name, a collection of employees, and a collection of external contacts. Each employee has a name, a salary, and a collection of tasks. Some contacts are clients." Unlike theirs, our database does not include an additional `id` field. Instead, we use PostgreSQL's automatically generated

[1] I apologize for the use of one of the most boring database schemas imaginable and promise to do better in the future.

```
table departments with (oid:Int, name:String)
table employees with (oid:Int, dept:String, name:String, salary:Int)
table tasks with (oid:Int, employee:String, task:String)
table contacts with (oid:Int, dept:String, name:String, client:Bool)
```

Figure 5.1: Benchmark database schema, cf. Cheney et al. [2014c].

`oid` column to identify rows in where-provenance and lineage. We populate the databases at varying sizes using randomly generated data in the same way: "We vary the number of departments in the organization from 4 to 4096 (by powers of 2). Each department has on average 100 employees and each employee has 0–2 tasks." The largest database, with 4096 departments, is 142 MB on disk when exported by `pg_dump` to a SQL file (which excludes the `oid` column). We create additional indices on `tasks(employee)`, `tasks(task)`, `employees(dept)`, and `contacts(dept)`.

All tests were performed on an otherwise idle desktop system with a 3.2 GHz quad-core CPU, 8 GB RAM, and a 500 GB HDD. The system ran Linux (kernel version 4.5.0) and we used PostgreSQL 9.4.2 as the database engine. Links and its variants Links^{W} and Links^{L} are interpreters written in OCaml, which were compiled to native code using OCaml 4.02.3. The exact versions of Links^{W} and Links^{L} used for this set of benchmarks can be downloaded here:

`https://www.inf.ed.ac.uk/research/isdd/admin/package?download=188`

`https://www.inf.ed.ac.uk/research/isdd/admin/package?download=189`

5.1 Links^{W}

To be usable in practice, where-provenance should not have unreasonable runtime overhead. The nature of where-provenance suggests that the cost of where-provenance annotations is linear in the size of the result. More precisely, if every single piece of data is annotated with a similarly sized provenance triple, we expect the runtime of a fully where-provenance-annotated query to be around four times the runtime of an unannotated query just for handling more data.

In the following set of benchmarks, we compare queries *without* any where-provenance against queries that calculate where-provenance on *some* of the result and queries that calculate *full* where-provenance wherever possible. This should give us an idea of the overhead of where-provenance on typical queries,

which are somewhere in between full and no provenance. The queries are based on those with nested results by Cheney et al. [2014c].

We only benchmark *default* where-provenance, that is table name, column name, and the database-generated `oid` for row identification. These are short strings and integers — just like the values in our test database. External provenance is computed by user-defined database-executable functions and can thus be arbitrarily expensive.

For *full* where-provenance we change the table declarations to add provenance to every field (except the `oid`). This changes the types, so we have to adapt the queries and some of the helper functions used inside the queries. Figure 5.2 shows the benchmark queries with *full* provenance and Figure 5.3 shows the helper functions. Note that, for example, query Q2 maps the **`data`** keyword over the employees tasks before comparing the tasks against *"abstract"*. Query Q6 returns the outliers in terms of salary and their tasks, concatenated with the clients, who are assigned the *fake* task *"buy"*. Since the fake task is not a database value it cannot have where-provenance and LINKS^W type system prevents us from pretending it does. Thus, the list of tasks has type `[String]`, not `[Prov(String)]`. Figures 5.4, 5.5, 5.6, and 5.7 show the SQL queries that are generated by LINKS for the queries with full where-provenance.

The queries with *some* where-provenance are derived from the queries with full provenance. Query Q1 drops provenance from the contacts' fields. Q2 returns data and provenance separately. It does not actually return less information, it is just less type-safe. Q3 drops provenance from the employee. Q4 returns the employees' provenance only, and drops the actual data. Q5 does not return provenance on the employees' fields. Q6 drops provenance on the department.

5.1.1 Setup

We have three LINKS^W programs, one for each level of where-provenance annotations. For each database size, we drop all tables and load a dump from disk, starting with 4096. We then run LINKS^W three times, once for each program in order *all*, *some*, *none*. Each of the three programs runs its queries 5 times in a row and reports the median runtime in milliseconds. The programs measure runtime using the LINKS^W built-in function `serverTimeMilliseconds`

```
sig q1: () -> [(name:Prov(String),
                contacts:[(client:Prov(Bool), name:Prov(String))],
                employees:[(name:Prov(String), salary:Prov(Int),
                            tasks:[Prov(String)])])]
fun q1() { for (d <-- departments)
  [(contacts = contactsOfDept(d),
    employees = employeesOfDept(d), name = d.name)] }

sig q2: () -> [(d: Prov(String))]
fun q2() { for (d <- q1()) where (all(d.employees, fun (e) {
     contains(map(fun (x) { data x }, e.tasks), "abstract") }))
  [(d = d.name)] }

sig q3: () -> [(b: [Prov(String)], e: Prov(String))]
fun q3() { for (e <-- employees) [(b = tasksOfEmp(e), e = e.name)] }

sig q4: () -> [(dpt:Prov(String), emps:[Prov(String)])]
fun q4 () { for (d <-- departments)
  [(dpt = d.name, emps = for (e <-- employees)
                         where ((data d.name) == (data e.dept))
                           [(e.name)])] }

sig q5: () -> [(a:Prov(String),
       b:[(name:Prov(String),salary:Prov(Int),tasks:[Prov(String)])])]
fun q5() { for (t <-- tasks) [(a = t.task, b = employeesByTask(t))] }

sig q6: () -> [(d:Prov(String),p:[(name:Prov(String),tasks:[String])])]
fun q6() { for (x <- q1())
  [(d = x.name, p = get(outliers(x.employees),
                        fun (y) { map(fun (z) { data z }, y.tasks) })
                ++ get(clients(x.contacts), fun (y) { ["buy"] }))]}
```

Figure 5.2: Links^{W} benchmark queries variant *allprov*.

```
fun tasksOfEmp(e) {
  for (t <-- tasks) where ((data t.employee) == data e.name)
    [t.task] }
fun contactsOfDept(d) {
  for (c <-- contacts) where ((data d.name) == data c.dept)
    [(client = c.client, name = c.name)] }
fun employeesByTask(t) {
  for (e <-- employees) for (d <-- departments)
  where ((data e.name) == (data t.employee)
      && (data e.dept) == (data d.name))
    [(name = e.name, salary = e.salary, tasks = tasksOfEmp(e))] }
fun employeesOfDept(d) {
  for (e <-- employees) where ((data d.name) == data e.dept)
    [(name = e.name, salary = e.salary, tasks = tasksOfEmp(e))] }
fun get(xs, f) { for (x <- xs) [(name = x.name, tasks = f(x))] }
fun outliers(xs) { filter(fun (x) { isRich(x) || isPoor(x) }, xs) }
fun clients(xs) { filter(fun (x) { data x.client }, xs) }
```

Figure 5.3: Helper functions variant *allprov*.

which in turn uses OCAML's `gettimeofday`. These measurements include query normalization time, running the actual generated query, and building the result, but not the time it takes to translate LINKS^W to LINKS.

5.1.2 Data

Figure 5.8 shows our experimental results. We have one plot for every query, showing the database size on the x-axis and the median runtime over five runs on the y-axis. Note that both axes are logarithmic. Measurements of *full* where-provenance are in black circles, *some* in blue squares, and *none* in yellow triangles. Based on test runs we had to exclude some results for queries at larger database sizes because the queries returned results that were too large for LINKS to construct as in-memory values. We discuss this flaw in the experimental design in Section 5.4. In short, there are ways to reduce the memory footprint, but the more typical use of provenance is debugging smaller results.

The graph for query Q2 looks a bit odd. This seems to be due to Q2 not actually returning any data for some database sizes, because for some of the (randomly generated) instances there just are no departments where all employees

```
SELECT 0 AS "1_1", t2061."2" AS "1_2", 4 AS "2_contacts_1",
       row_number() OVER (ORDER BY t2061."2", t2049.name) AS "2_contacts_2",
       4 AS "2_employees_1",
       row_number() OVER (ORDER BY t2061."2", t2049.name) AS "2_employees_2",
       t2049.name AS "2_name_!data", 'departments' AS "2_name_!prov_1",
       'name' AS "2_name_!prov_2", t2049.oid AS "2_name_!prov_3"
  FROM (SELECT 1 AS "2") AS t2061, departments AS t2049;

SELECT 4 AS "1_1", t2063."2" AS "1_2", t2057.client AS "2_client_!data",
       'contacts' AS "2_client_!prov_1", 'client' AS "2_client_!prov_2",
       t2057.oid AS "2_client_!prov_3", t2057.name AS "2_name_!data",
       'contacts' AS "2_name_!prov_1", 'name' AS "2_name_!prov_2",
       t2057.oid AS "2_name_!prov_3"
  FROM (SELECT t2049.name AS "1_1_name", t2049.oid AS "1_1_oid",
               row_number() OVER (ORDER BY t2049.name) AS "2"
          FROM departments AS t2049) AS t2063,
       contacts AS t2057
 WHERE t2063."1_1_name" = t2057.dept;

SELECT 4 AS "1_1", t2065."2" AS "1_2", t2058.name AS "2_name_!data",
       'employees' AS "2_name_!prov_1", 'name' AS "2_name_!prov_2",
       t2058.oid AS "2_name_!prov_3", t2058.salary AS "2_salary_!data",
       'employees' AS "2_salary_!prov_1", 'salary' AS "2_salary_!prov_2",
       t2058.oid AS "2_salary_!prov_3", 3 AS "2_tasks_1",
       row_number() OVER (ORDER BY t2065."2", t2058.name) AS "2_tasks_2"
  FROM (SELECT t2049.name AS "1_1_name", t2049.oid AS "1_1_oid",
               row_number() OVER (ORDER BY t2049.name) AS "2"
          FROM departments AS t2049) AS t2065,
       employees AS t2058
 WHERE t2065."1_1_name" = t2058.dept;

SELECT 3 AS "1_1", t2067."2" AS "1_2", t2059.task AS "2_!data",
       'tasks' AS "2_!prov_1", 'task' AS "2_!prov_2", t2059.oid AS "2_!prov_3"
  FROM (SELECT t2049.name AS "1_1_name", t2049.oid AS "1_1_oid",
               t2058.dept AS "1_2_dept", t2058.name AS "1_2_name",
               t2058.oid AS "1_2_oid", t2058.salary AS "1_2_salary",
               row_number() OVER (ORDER BY t2049.name, t2058.name) AS "2"
          FROM departments AS t2049, employees AS t2058
         WHERE t2049.name = t2058.dept) AS t2067,
       tasks AS t2059
 WHERE t2059.employee = t2067."1_2_name";
```

Figure 5.4: Generated where-provenace query Q1 in variant *allprov*.

```
-- Q2
SELECT 0 AS "1_1", t2209."2" AS "1_2", t2160.name AS "2_d_!data",
       'departments' AS "2_d_!prov_1", 'name' AS "2_d_!prov_2",
       t2160.oid AS "2_d_!prov_3"
  FROM (SELECT 1 AS "2") AS t2209, departments AS t2160
 WHERE NOT ( EXISTS (
         SELECT 0 AS dummy
           FROM employees AS t2203
          WHERE t2160.name = t2203.dept
            AND (NOT (EXISTS (SELECT 0 AS dummy
                                FROM tasks AS t2204
                               WHERE t2204.employee = t2203.name
                                 AND t2204.task = ('abstract'))))));

-- Q3
SELECT 0 AS "1_1", t2464."2" AS "1_2", 2 AS "2_b_1",
       row_number() OVER (ORDER BY t2464."2", t2459.name) AS "2_b_2",
       t2459.name AS "2_e_!data", 'employees' AS "2_e_!prov_1",
       'name' AS "2_e_!prov_2", t2459.oid AS "2_e_!prov_3"
  FROM (SELECT 1 AS "2") AS t2464, employees AS t2459;

SELECT 2 AS "1_1", t2466."2" AS "1_2", t2462.task AS "2_!data",
       'tasks' AS "2_!prov_1", 'task' AS "2_!prov_2",
       t2462.oid AS "2_!prov_3"
  FROM (SELECT t2459.dept AS "1_1_dept", t2459.name AS "1_1_name",
               t2459.oid AS "1_1_oid", t2459.salary AS "1_1_salary",
               row_number() OVER (ORDER BY t2459.name) AS "2"
          FROM employees AS t2459) AS t2466,
       tasks AS t2462
 WHERE t2462.employee = t2466."1_1_name";
```

Figure 5.5: Generated where-provenace queries Q2 and Q3 in variant *allprov*.

```
-- Q4
SELECT 0 AS "1_1", t2509."2" AS "1_2", t2504.name AS "2_dpt_!data",
       'departments' AS "2_dpt_!prov_1", 'name' AS "2_dpt_!prov_2",
       t2504.oid AS "2_dpt_!prov_3", 2 AS "2_emps_1",
       row_number() OVER (ORDER BY t2509."2", t2504.name) AS "2_emps_2"
  FROM (SELECT 1 AS "2") AS t2509, departments AS t2504;

SELECT 2 AS "1_1", t2511."2" AS "1_2", t2507.name AS "2_!data",
       'employees' AS "2_!prov_1", 'name' AS "2_!prov_2", t2507.oid AS "2_!prov_3"
  FROM (SELECT t2504.name AS "1_1_name", t2504.oid AS "1_1_oid",
               row_number() OVER (ORDER BY t2504.name) AS "2"
          FROM departments AS t2504 ) AS t2511, employees AS t2507
 WHERE t2511."1_1_name" = t2507.dept;

-- Q5
SELECT 0 AS "1_1", t2561."2" AS "1_2", t2549.task AS "2_a_!data",
       'tasks' AS "2_a_!prov_1", 'task' AS "2_a_!prov_2",
       t2549.oid AS "2_a_!prov_3", 3 AS "2_b_1",
       row_number() OVER (ORDER BY t2561."2", t2549.oid) AS "2_b_2"
  FROM (SELECT 1 AS "2") AS t2561, tasks AS t2549;

SELECT 3 AS "1_1", t2563."2" AS "1_2", t2557.name AS "2_name_!data",
       'employees' AS "2_name_!prov_1", 'name' AS "2_name_!prov_2",
       t2557.oid AS "2_name_!prov_3", t2557.salary AS "2_salary_!data",
       'employees' AS "2_salary_!prov_1", 'salary' AS "2_salary_!prov_2",
       t2557.oid AS "2_salary_!prov_3", 2 AS "2_tasks_1",
       row_number() OVER (ORDER BY t2563."2", t2557.name, t2558.name) AS "2_tasks_2"
  FROM (SELECT t2549.employee AS "1_1_employee", t2549.oid AS "1_1_oid",
               t2549.task AS "1_1_task",
               row_number() OVER (ORDER BY t2549.oid) AS "2"
          FROM tasks AS t2549) AS t2563,
       employees AS t2557, departments AS t2558
 WHERE t2557.name = t2563."1_1_employee" AND t2557.dept = t2558.name;

SELECT 2 AS "1_1", t2565."2" AS "1_2", t2559.task AS "2_!data",
       'tasks' AS "2_!prov_1", 'task' AS "2_!prov_2", t2559.oid AS "2_!prov_3"
  FROM (SELECT t2549.employee AS "1_1_employee", t2549.oid AS "1_1_oid",
               t2549.task AS "1_1_task", t2557.dept AS "1_2_dept",
               t2557.name AS "1_2_name", t2557.oid AS "1_2_oid",
               t2557.salary AS "1_2_salary", t2558.name AS "1_3_name",
               t2558.oid AS "1_3_oid", row_number() OVER (
                ORDER BY t2549.oid, t2557.name, t2558.name) AS "2"
          FROM tasks AS t2549, employees AS t2557, departments AS t2558
         WHERE t2557.name = t2549.employee AND t2557.dept = t2558.name
       ) AS t2565, tasks AS t2559 WHERE t2559.employee = t2565."1_2_name";
```

Figure 5.6: Generated where-provenace queries Q4 and Q5 in variant *allprov*.

```
SELECT 0 AS "1_1", t2711."2" AS "1_2", t2650.name AS "2_department_!data",
       'departments' AS "2_department_!prov_1", 'name' AS "2_department_!prov_2",
       t2650.oid AS "2_department_!prov_3", 5 AS "2_people_1",
       row_number() OVER (ORDER BY t2711."2", t2650.name) AS "2_people_2"
  FROM (SELECT 1 AS "2") AS t2711, departments AS t2650;

  (SELECT 5 AS "1_1", t2713."2" AS "1_2", t2707.name AS "2_name_!data",
          'employees' AS "2_name_!prov_1", 'name' AS "2_name_!prov_2",
          t2707.oid AS "2_name_!prov_3", 2 AS "2_tasks_1",
          row_number() OVER (ORDER BY t2713."2", t2707.name) AS "2_tasks_2"
     FROM (SELECT t2650.name AS "1_1_name", t2650.oid AS "1_1_oid",
                  row_number() OVER (ORDER BY t2650.name) AS "2"
             FROM departments AS t2650) AS t2713, employees AS t2707
    WHERE t2713."1_1_name" = t2707.dept
      AND (t2707.salary > (1000000) OR t2707.salary < (1000)))
 UNION ALL (SELECT 5 AS "1_1", t2715."2" AS "1_2",
                   t2709.name AS "2_name_!data", 'contacts' AS "2_name_!prov_1",
                   'name' AS "2_name_!prov_2", t2709.oid AS "2_name_!prov_3",
                   4 AS "2_tasks_1", row_number() OVER (
                    ORDER BY t2715."2", t2709.name) AS "2_tasks_2"
              FROM (SELECT t2650.name AS "1_1_name", t2650.oid AS "1_1_oid",
                           row_number() OVER (ORDER BY t2650.name) AS "2"
                      FROM departments AS t2650) AS t2715,
                   contacts AS t2709
             WHERE t2715."1_1_name" = t2709.dept AND t2709.client);

  (SELECT 2 AS "1_1", t2717."2" AS "1_2", t2708.task AS "2"
     FROM (SELECT t2650.name AS "1_1_name", t2650.oid AS "1_1_oid",
                  t2707.dept AS "1_2_dept", t2707.name AS "1_2_name",
                  t2707.oid AS "1_2_oid", t2707.salary AS "1_2_salary",
                  row_number() OVER (ORDER BY t2650.name, t2707.name) AS "2"
             FROM departments AS t2650, employees AS t2707
            WHERE t2650.name = t2707.dept
              AND (t2707.salary > (1000000) OR t2707.salary < (1000))
          ) AS t2717, tasks AS t2708
    WHERE t2708.employee = t2717."1_2_name")
UNION ALL (SELECT 4 AS "1_1", t2719."2" AS "1_2", 'buy' AS "2"
           FROM (SELECT t2650.name AS "1_1_name", t2650.oid AS "1_1_oid",
                        t2709.client AS "1_2_client", t2709.dept AS "1_2_dept",
                        t2709.name AS "1_2_name", t2709.oid AS "1_2_oid",
                        row_number() OVER (
                         ORDER BY t2650.name, t2709.name ) AS "2"
                   FROM departments AS t2650, contacts AS t2709
                  WHERE t2650.name = t2709.dept AND t2709.client) AS t2719);
```

Figure 5.7: Generated where-provenace query Q6 in variant *allprov*.

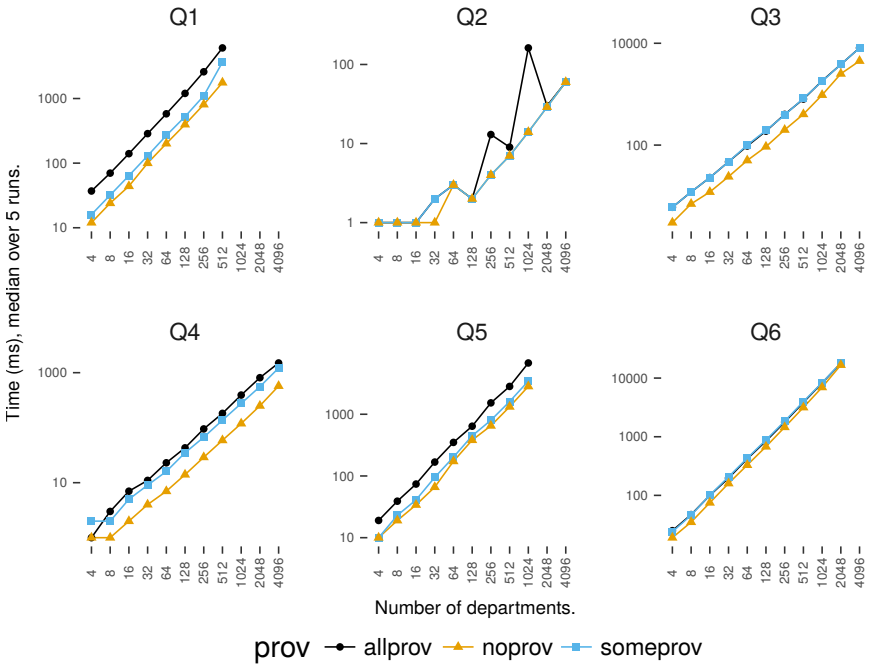

Figure 5.8: Where-provenance query runtimes.

Query	#depts	runtime in ms			overall slowdown
		allprov	someprov	noprov	(geom mean)
Q1	512	6068	3653	1763	2.26
Q2	4096	60	60	60	1.52
Q3	4096	8100	8064	4497	1.88
Q4	4096	1502	1214	573	2.80
Q5	1024	6778	3457	2832	1.85
Q6	2048	17874	18092	16716	1.22

Figure 5.9: Median runtimes for largest data set a query ran on and geometric means of overall slowdown across all instance sizes.

have the task *"abstract"*.

The table in Figure 5.9 shows the median runtime in milliseconds of every query on the largest database instance. For most queries this is 4096; for Q1 it is 512, 1024 for Q5, and 2048 for Q6. These are the values of the right-most data points for every variant in every query's graph above.

The table also lists, for every query, the slowdown of full where-provenance versus no provenance as the geometric mean across all database sizes. The slowdown ranges from 1.22 for query Q6 up to 2.8 for query Q4. Note that query Q2 has the same runtime for all variants at 4096 departments, but full provenance is slower for some database sizes, so the overall slowdown is > 1.

5.1.3 Interpretation

The graphs suggest that indeed the overhead of where-provenance is indeed linear in the result size as the lines are more or less parallel. This was expected, anything else would have suggested a bug in the implementation.

The multiplicative overhead seems to be larger for queries that return more data. Notably, for query Q2, which returns no data at all on some of our test database instances, the overhead is hardly visible. The raw amount of data returned for the full where-provenance queries is three to four times that of a plain query. Most strings are short names and provenance adds two short strings and a number for table, column, and row. The largest overhead is 2.8 for query Q4. This is less than 4, so Links^{W} exceeds our expectations due to just raw additional data needing to be processed. One possible explanation is that the where-provenance in our benchmark queries is largely static; there are only four tables with a maximum of four columns each.

5.2 Links^{L}

We expect lineage to have different performance characteristics than where-provenance. Unlike where-provenance, lineage is conceptually set valued. A query with few actual results could have huge lineage, because lineage is combined for equal data. In practice, due to Links using multiset semantics for queries, the amount of lineage is bounded by the shape of the query. Thus, we expect lineage queries to have the same asymptotic cost as queries without lineage.

However, the lineage translation still replaces single comprehensions by nested comprehensions that combine lineage. We expect this to have a larger impact on performance than where-provenance, where we only needed to trace a little more data through the query.

Mea culpa: I made a mistake in the previously published versions of this set of benchmarks [Fehrenbach and Cheney, 2016, 2018]. I would like to thank Seokki Lee for finding this mistake when benchmarking Links^{L} against Pug [Lee et al., 2018] and pointing it out to me. Query Q7 had excessive runtimes due to a mistake I made when inlining a helper function by hand because the Links^{L} prototype does not handle functions correctly. Instead of joining departments and employees who are outliers with respect to their salary, it joined matching employees with big salaries and all employees with small salaries. We reran the set of benchmarks described in this section on the same hardware, but using a more recent version of PostgreSQL, namely 10.5. Comparing the correct results here to the previously published results, we see that almost all queries ran faster. While this caused the overhead of lineage to be bigger than previously reported for some queries, it also demonstrates that language-integrated provenance makes it easy to benefit from continuing improvements to database systems.

5.2.1 Setup

Figure 5.10 lists the queries used in the lineage experiments. To compute lineage, queries are wrapped in a **`lineage`** block. The prototype implementation of Links^{L} does not currently handle function calls in lineage blocks correctly, so in our experiments we used manually written lineage-enabled versions of the functions `employeesByTask` and `tasksOfEmp`, whose bodies are wrapped in a **`lineage`** block. This results in the same Sql query a correct implementation would produce automatically. It has a negligible effect on normalization time. We reuse some of the queries from the where-provenance experiments, namely Q3, Q4, and Q5. Queries AQ6, Q6N, and Q7 are inspired by query Q6, but not quite the same. Queries QF3 and QF4 are two of the flat queries used by Cheney et al. [2014c]. Query QC4 computes pairs of employees in the same department and their tasks in a "tagged union". Figures 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.19, 5.20, 5.21, 5.22, and 5.23 show the generated Sql queries.

We use a similar experimental setup to the one for where-provenance. We

```
fun aq6() { for (d <- for (d <-- departments)
    [(employees=for (e <-- employees) where (d.name == e.dept)
                  [(name=e.name, salary=e.salary)], name=d.name)])
  [(department=d.name,
    outliers=for (o <- d.employees)
             where (o.salary > 1000000 || o.salary < 1000) [o])]}
fun q3() { for (e <-- employees) [(b=tasksOfEmp(e), e=e.name)]}
fun q4() { for (d <-- departments)
  [(dpt=d.name,
    emps=for (e <-- employees) where (d.name == e.dept) [e.name])]}
fun q5() { for (t <-- tasks) [(a=t.task, b=employeesByTask(t))]}
fun q6n() { for (x <-- departments) [(department=x.name, people=
  (for (y <-- employees)
   where (x.name==y.dept && (y.salary<1000 || y.salary>1000000))
     [(name=y.name, tasks=for (z <-- tasks)
                          where (z.employee == y.name) [z.task])])
   ++ (for (y <-- contacts) where (x.name == y.dept && y.client)
         [(name=y.dept, tasks=["buy"])]))]}
fun q7() { for (d <-- departments) for (e <-- employees)
  where (d.name == e.dept && (e.salary > 1000000 || e.salary < 1000))
    [(employee=(name=e.name, salary=e.salary), department=d.name)]}
fun qc4() { for (x <-- employees) for (y <-- employees)
          where (x.dept == y.dept && x.name <> y.name)
  [(a=x.name, b=y.name,
    c=(for (t <-- tasks) where (x.name == t.employee)
         [(doer="a", task=t.task)])
       ++ (for (t <-- tasks) where (y.name == t.employee)
             [(doer="b", task=t.task)]))]}
fun qf3() { for (e <-- employees) for (f <-- employees)
  where (e.dept==f.dept && e.salary==f.salary && e.name<>f.name)
    [(e.name, f.name)]}
var QF4 =
  (for (t <-- tasks) where (t.task == "abstract") [t.employee]) ++
  (for (e <-- employees) where (e.salary > 50000) [e.name])}
```

Figure 5.10: Lineage queries used in experiments.

```
fun employeesByTask(t) { lineage {
  for (e <-- employees) for (d <-- departments)
  where (e.name == t.employee && e.dept == d.name)
    [(name = e.name, salary = e.salary, tasks = tasksOfEmp(e))]}}

fun tasksOfEmp(e) { lineage {
  for (t <-- tasks) where (t.employee == e.name) [t.task] } }
```

Figure 5.11: Lineage-enabled helper functions.

only use databases up to 1024 departments, because most of the queries are a lot more expensive. Query QC4 computes pairs of employees and their tasks, which has at least quadratic complexity. It has excessive runtime even for very small databases. We excluded it from runs on larger databases.

5.2.2 Data

Figure 5.24 shows our lineage experiment results. Again, we have one plot for every query, showing the database size on the x-axis and the median runtime over five runs on the y-axis. Both axes are logarithmic. Measurements with lineage are in black circles, no lineage is shown as yellow triangles.

The table in Figure 5.25 lists queries and their median runtimes with and without lineage. The time reported is in milliseconds, for the largest database instance that both variants of a query ran on. For most queries this is 1024; but it is only 32 for QC4. The table also reports the slowdown of lineage versus no lineage as the geometric mean over all database sizes. The performance penalty for using lineage ranges from query QC4 needing 80 percent more time to query QF3 being more than 8 times slower than its counterpart.

5.2.3 Interpretation

The experiments confirm that lineage has more or less linear overhead. Lineage is still somewhat expensive to compute, with slowdowns ranging from 1.8 to more than 8 times slower. At the moment we do not have a clear idea what causes greater slowdowns. We can rule out nesting depth, seeing that Q4 and Q5 have the same but are at opposite ends of the slowdown range and the flat

```
SELECT 0 AS "1_1", t2499."2" AS "1_2", t2485.name AS "2_data_department",
       4 AS "2_data_outliers_1",
       row_number() OVER (ORDER BY t2499."2", t2485.name) AS "2_data_outliers_2",
       4 AS "2_prov_1",
       row_number() OVER (ORDER BY t2499."2", t2485.name) AS "2_prov_2"
  FROM (SELECT 1 AS "2") AS t2499, departments AS t2485;

SELECT 4 AS "1_1", t2501."2" AS "1_2", t2495.name AS "2_data_name",
       t2495.salary AS "2_data_salary", 2 AS "2_prov_1",
       row_number() OVER (ORDER BY t2501."2", t2495.name) AS "2_prov_2"
  FROM (SELECT t2485.name AS "1_1_name", t2485.oid AS "1_1_oid",
               row_number() OVER (ORDER BY t2485.name) AS "2"
          FROM departments AS t2485) AS t2501, employees AS t2495
 WHERE t2501."1_1_name" = t2495.dept
   AND (t2495.salary > (1000000) OR t2495.salary < (1000));

SELECT 2 AS "1_1", t2503."2" AS "1_2", t2503."1_2_oid" AS "2_row",
       'employees' AS "2_table"
  FROM (SELECT t2485.name AS "1_1_name", t2485.oid AS "1_1_oid",
               t2495.dept AS "1_2_dept", t2495.name AS "1_2_name",
               t2495.oid AS "1_2_oid", t2495.salary AS "1_2_salary",
               row_number() OVER (ORDER BY t2485.name, t2495.name) AS "2"
          FROM departments AS t2485, employees AS t2495
         WHERE t2485.name = t2495.dept
          AND (t2495.salary > (1000000) OR t2495.salary < (1000))
       ) AS t2503;

SELECT 4 AS "1_1", t2505."2" AS "1_2", t2505."1_1_oid" AS "2_row",
       'departments' AS "2_table"
  FROM (SELECT t2485.name AS "1_1_name", t2485.oid AS "1_1_oid",
               row_number() OVER (ORDER BY t2485.name) AS "2"
          FROM departments AS t2485) AS t2505;
```

Figure 5.12: Generated lineage query AQ6.

```
SELECT 0 AS "1_1", t2404."2" AS "1_2", 4 AS "2_data_b_1",
       row_number() OVER (ORDER BY t2404."2", t2395.name) AS "2_data_b_2",
       t2395.name AS "2_data_e", 4 AS "2_prov_1",
       row_number() OVER (ORDER BY t2404."2", t2395.name) AS "2_prov_2"
  FROM (SELECT 1 AS "2") AS t2404, employees AS t2395;

SELECT 4 AS "1_1", t2406."2" AS "1_2", t2401.task AS "2_data",
       2 AS "2_prov_1",
       row_number() OVER (ORDER BY t2406."2", t2401.oid) AS "2_prov_2"
  FROM (SELECT t2395.dept AS "1_1_dept", t2395.name AS "1_1_name",
               t2395.oid AS "1_1_oid", t2395.salary AS "1_1_salary",
               row_number() OVER (ORDER BY t2395.name) AS "2"
          FROM employees AS t2395) AS t2406, tasks AS t2401
 WHERE t2401.employee = t2406."1_1_name";

SELECT 2 AS "1_1", t2408."2" AS "1_2", t2408."1_2_oid" AS "2_row",
       'tasks' AS "2_table"
  FROM (SELECT t2395.dept AS "1_1_dept", t2395.name AS "1_1_name",
               t2395.oid AS "1_1_oid", t2395.salary AS "1_1_salary",
               t2401.employee AS "1_2_employee", t2401.oid AS "1_2_oid",
               t2401.task AS "1_2_task",
               row_number() OVER (ORDER BY t2395.name, t2401.oid) AS "2"
          FROM employees AS t2395, tasks AS t2401
         WHERE t2401.employee = t2395.name ) AS t2408;

SELECT 4 AS "1_1", t2410."2" AS "1_2", t2410."1_1_oid" AS "2_row",
       'employees' AS "2_table"
  FROM (SELECT t2395.dept AS "1_1_dept", t2395.name AS "1_1_name",
               t2395.oid AS "1_1_oid", t2395.salary AS "1_1_salary",
               row_number() OVER (ORDER BY t2395.name) AS "2"
          FROM employees AS t2395) AS t2410;
```

Figure 5.13: Generated lineage query Q3.

```
SELECT 0 AS "1_1", t2421."2" AS "1_2", t2412.name AS "2_data_dpt",
       4 AS "2_data_emps_1",
       row_number() OVER (ORDER BY t2421."2", t2412.name) AS "2_data_emps_2",
       4 AS "2_prov_1",
       row_number() OVER (ORDER BY t2421."2", t2412.name) AS "2_prov_2"
  FROM (SELECT 1 AS "2") AS t2421, departments AS t2412;

SELECT 4 AS "1_1", t2423."2" AS "1_2", t2418.name AS "2_data",
       2 AS "2_prov_1",
       row_number() OVER (ORDER BY t2423."2", t2418.name) AS "2_prov_2"
  FROM (SELECT t2412.name AS "1_1_name", t2412.oid AS "1_1_oid",
               row_number() OVER (ORDER BY t2412.name) AS "2"
          FROM departments AS t2412 ) AS t2423, employees AS t2418
 WHERE t2423."1_1_name" = t2418.dept;

SELECT 2 AS "1_1", t2425."2" AS "1_2", t2425."1_2_oid" AS "2_row",
       'employees' AS "2_table"
  FROM (SELECT t2412.name AS "1_1_name", t2412.oid AS "1_1_oid",
               t2418.dept AS "1_2_dept", t2418.name AS "1_2_name",
               t2418.oid AS "1_2_oid", t2418.salary AS "1_2_salary",
               row_number() OVER (ORDER BY t2412.name, t2418.name) AS "2"
          FROM departments AS t2412, employees AS t2418
         WHERE t2412.name = t2418.dept ) AS t2425;

SELECT 4 AS "1_1", t2427."2" AS "1_2", t2427."1_1_oid" AS "2_row",
       'departments' AS "2_table"
  FROM (SELECT t2412.name AS "1_1_name", t2412.oid AS "1_1_oid",
               row_number() OVER (ORDER BY t2412.name) AS "2"
          FROM departments AS t2412 ) AS t2427;
```

Figure 5.14: Generated lineage query Q4.

```
SELECT 0 AS "1_1", t2463."2" AS "1_2", t2429.task AS "2_data_a", 7 AS "2_data_b_1",
  row_number() OVER (ORDER BY t2463."2", t2429.oid) AS "2_data_b_2",
  7 AS "2_prov_1", row_number() OVER (ORDER BY t2463."2", t2429.oid) AS "2_prov_2"
FROM (SELECT 1 AS "2") AS t2463, tasks AS t2429;

SELECT 7 AS "1_1", t2465."2" AS "1_2", t2456.name AS "2_data_name", t2456.salary
  AS "2_data_salary", 5 AS "2_data_tasks_1", row_number() OVER (ORDER BY
  t2465."2", t2456.name, t2457.name ) AS "2_data_tasks_2", 5 AS "2_prov_1",
  row_number() OVER (ORDER BY t2465."2", t2456.name, t2457.name) AS "2_prov_2"
FROM (SELECT t2429.employee AS "1_1_employee", t2429.oid AS "1_1_oid",
             t2429.task AS "1_1_task",
             row_number() OVER (ORDER BY t2429.oid) AS "2"
        FROM tasks AS t2429) AS t2465, employees AS t2456, departments AS t2457
 WHERE t2456.name = t2465."1_1_employee" AND t2456.dept = t2457.name;

SELECT 5 AS "1_1", t2467."2" AS "1_2", t2458.task AS "2_data", 2 AS "2_prov_1",
       row_number() OVER (ORDER BY t2467."2", t2458.oid) AS "2_prov_2"
  FROM (SELECT t2429.employee AS "1_1_employee", t2429.oid AS "1_1_oid",
               t2429.task AS "1_1_task", t2456.dept AS "1_2_dept",
               t2456.name AS "1_2_name", t2456.oid AS "1_2_oid",
               t2456.salary AS "1_2_salary", t2457.name AS "1_3_name",
               t2457.oid AS "1_3_oid",
               row_number() OVER (ORDER BY t2429.oid, t2456.name, t2457.name) AS "2"
          FROM tasks AS t2429, employees AS t2456, departments AS t2457
         WHERE t2456.name = t2429.employee AND t2456.dept = t2457.name) AS t2467,
       tasks AS t2458 WHERE t2458.employee = t2467."1_2_name";

SELECT 2 AS "1_1", t2469."2" AS "1_2", t2469."1_4_oid" AS "2_row", 'tasks' AS "2_table"
  FROM (SELECT t2429.employee AS "1_1_employee", t2429.oid AS "1_1_oid",
               t2429.task AS "1_1_task", t2456.dept AS "1_2_dept",
               t2456.name AS "1_2_name", t2456.oid AS "1_2_oid",
               t2456.salary AS "1_2_salary", t2457.name AS "1_3_name",
               t2457.oid AS "1_3_oid", t2458.employee AS "1_4_employee",
               t2458.oid AS "1_4_oid", t2458.task AS "1_4_task",
               row_number() OVER (
                ORDER BY t2429.oid, t2456.name, t2457.name, t2458.oid
               ) AS "2"
          FROM tasks AS t2429, employees AS t2456, departments AS t2457,
               tasks AS t2458
         WHERE     (t2456.name = t2429.employee AND t2456.dept = t2457.name)
               AND t2458.employee = t2456.name
       ) AS t2469;
```

Figure 5.15: Generated lineage query Q5 (part 1).

```
(SELECT 5 AS "1_1", t2471."2" AS "1_2", t2471."1_2_oid" AS "2_row",
        'employees' AS "2_table"
   FROM (SELECT t2429.employee AS "1_1_employee", t2429.oid AS "1_1_oid",
                t2429.task AS "1_1_task", t2456.dept AS "1_2_dept",
                t2456.name AS "1_2_name", t2456.oid AS "1_2_oid",
                t2456.salary AS "1_2_salary", t2457.name AS "1_3_name",
                t2457.oid AS "1_3_oid",
                row_number() OVER (
                 ORDER BY t2429.oid, t2456.name, t2457.name) AS "2"
           FROM tasks AS t2429, employees AS t2456, departments AS t2457
          WHERE t2456.name = t2429.employee AND t2456.dept = t2457.name
        ) AS t2471) UNION ALL
(SELECT 5 AS "1_1", t2473."2" AS "1_2",
        t2473."1_3_oid" AS "2_row", 'departments' AS "2_table"
   FROM (SELECT t2429.employee AS "1_1_employee", t2429.oid AS "1_1_oid",
                t2429.task AS "1_1_task", t2456.dept AS "1_2_dept",
                t2456.name AS "1_2_name", t2456.oid AS "1_2_oid",
                t2456.salary AS "1_2_salary", t2457.name AS "1_3_name",
                t2457.oid AS "1_3_oid", row_number() OVER (
                 ORDER BY t2429.oid, t2456.name, t2457.name ) AS "2"
           FROM tasks AS t2429, employees AS t2456, departments AS t2457
          WHERE t2456.name = t2429.employee
            AND t2456.dept = t2457.name) AS t2473);

SELECT 7 AS "1_1",
       t2475."2" AS "1_2",
       t2475."1_1_oid" AS "2_row",
       'tasks' AS "2_table"
  FROM (
        SELECT t2429.employee AS "1_1_employee",
               t2429.oid AS "1_1_oid",
               t2429.task AS "1_1_task",
               row_number() OVER (ORDER BY t2429.oid) AS "2"
          FROM tasks AS t2429
       ) AS t2475;
```

Figure 5.16: Generated lineage query Q5 (part 2).

```
SELECT 0 AS "1_1", t2614."2" AS "1_2", t2586.name AS "2_data_department",
       9 AS "2_data_people_1",
       row_number() OVER (ORDER BY t2614."2", t2586.name) AS "2_data_people_2",
       9 AS "2_prov_1",
       row_number() OVER (ORDER BY t2614."2", t2586.name) AS "2_prov_2"
  FROM (SELECT 1 AS "2") AS t2614, departments AS t2586;

(SELECT 9 AS "1_1", t2616."2" AS "1_2", t2607.name AS "2_data_name",
        4 AS "2_data_tasks_1",
        row_number() OVER (ORDER BY t2616."2", t2607.name) AS "2_data_tasks_2",
        4 AS "2_prov_1",
        row_number() OVER (ORDER BY t2616."2", t2607.name) AS "2_prov_2"
   FROM (SELECT t2586.name AS "1_1_name", t2586.oid AS "1_1_oid",
                row_number() OVER (ORDER BY t2586.name) AS "2"
           FROM departments AS t2586) AS t2616, employees AS t2607
  WHERE     t2616."1_1_name" = t2607.dept
        AND (t2607.salary < (1000) OR t2607.salary > (1000000))
) UNION ALL (SELECT 9 AS "1_1", t2618."2" AS "1_2",
  t2609.dept AS "2_data_name", 7 AS "2_data_tasks_1", row_number() OVER ( ORDER BY
  t2618."2", t2609.name ) AS "2_data_tasks_2", 7 AS "2_prov_1", row_number() OVER
  ( ORDER BY t2618."2", t2609.name ) AS "2_prov_2"
 FROM (SELECT t2586.name AS "1_1_name", t2586.oid AS "1_1_oid",
              row_number() OVER (ORDER BY t2586.name) AS "2"
         FROM departments AS t2586 ) AS t2618, contacts AS t2609
  WHERE t2618."1_1_name" = t2609.dept AND t2609.client);

(SELECT 4 AS "1_1", t2620."2" AS "1_2", t2608.task AS "2_data", 2 AS "2_prov_1",
        row_number() OVER (ORDER BY t2620."2", t2608.oid) AS "2_prov_2"
   FROM (SELECT t2586.name AS "1_1_name", t2586.oid AS "1_1_oid",
                t2607.dept AS "1_2_dept", t2607.name AS "1_2_name",
                t2607.oid AS "1_2_oid", t2607.salary AS "1_2_salary",
                row_number() OVER (ORDER BY t2586.name, t2607.name) AS "2"
           FROM departments AS t2586, employees AS t2607
          WHERE     t2586.name = t2607.dept
                AND (t2607.salary < (1000) OR t2607.salary > (1000000))
        ) AS t2620, tasks AS t2608
  WHERE t2608.employee = t2620."1_2_name") UNION ALL
(SELECT 7 AS "1_1", t2622."2" AS "1_2", 'buy' AS "2_data", 5 AS "2_prov_1",
        row_number() OVER (ORDER BY t2622."2") AS "2_prov_2"
   FROM (SELECT t2586.name AS "1_1_name", t2586.oid AS "1_1_oid",
                t2609.client AS "1_2_client", t2609.dept AS "1_2_dept",
                t2609.name AS "1_2_name", t2609.oid AS "1_2_oid",
                row_number() OVER (ORDER BY t2586.name, t2609.name ) AS "2"
           FROM departments AS t2586, contacts AS t2609
          WHERE t2586.name = t2609.dept AND t2609.client ) AS t2622;
```

Figure 5.17: Generated lineage query Q6N (part 1).

```
SELECT 2 AS "1_1", t2624."2" AS "1_2", t2624."1_3_oid" AS "2_row", 'tasks' AS "2_table"
  FROM (SELECT t2586.name AS "1_1_name", t2586.oid AS "1_1_oid",
               t2607.dept AS "1_2_dept", t2607.name AS "1_2_name",
               t2607.oid AS "1_2_oid", t2607.salary AS "1_2_salary",
               t2608.employee AS "1_3_employee", t2608.oid AS "1_3_oid",
               t2608.task AS "1_3_task", row_number() OVER (ORDER BY
                  t2586.name, t2607.name, t2608.oid) AS "2"
          FROM departments AS t2586, employees AS t2607, tasks AS t2608
          WHERE (t2586.name = t2607.dept AND (t2607.salary < (1000) OR
            t2607.salary > (1000000))) AND t2608.employee = t2607.name
       ) AS t2624;

(SELECT 4 AS "1_1", t2626."2" AS "1_2", t2626."1_2_oid" AS "2_row",
        'employees' AS "2_table"
   FROM (SELECT t2586.name AS "1_1_name", t2586.oid AS "1_1_oid",
                t2607.dept AS "1_2_dept", t2607.name AS "1_2_name",
                t2607.oid AS "1_2_oid", t2607.salary AS "1_2_salary",
                row_number() OVER (ORDER BY t2586.name, t2607.name) AS "2"
           FROM departments AS t2586, employees AS t2607
          WHERE     t2586.name = t2607.dept
                AND (t2607.salary < (1000) OR t2607.salary > (1000000))
        ) AS t2626 ) UNION ALL
(SELECT 7 AS "1_1", t2628."2" AS "1_2",
        t2628."1_2_oid" AS "2_row", 'contacts' AS "2_table"
   FROM (SELECT t2586.name AS "1_1_name", t2586.oid AS "1_1_oid",
                t2609.client AS "1_2_client", t2609.dept AS "1_2_dept",
                t2609.name AS "1_2_name", t2609.oid AS "1_2_oid",
                row_number() OVER (
                 ORDER BY t2586.name, t2609.name ) AS "2"
           FROM departments AS t2586, contacts AS t2609
          WHERE t2586.name = t2609.dept AND t2609.client ) AS t2628);

SELECT 9 AS "1_1", t2630."2" AS "1_2", t2630."1_1_oid" AS "2_row",
       'departments' AS "2_table"
  FROM (SELECT t2586.name AS "1_1_name", t2586.oid AS "1_1_oid",
               row_number() OVER (ORDER BY t2586.name) AS "2"
          FROM departments AS t2586 ) AS t2630;
```

Figure 5.18: Generated lineage query Q6N (part 2).

```
SELECT 0 AS "1_1", t2512."2" AS "1_2", t2507.name AS "2_data_department",
       t2510.name AS "2_data_employee_name", t2510.salary AS "2_data_employee_salary",
       3 AS "2_prov_1",
       row_number() OVER (ORDER BY t2512."2", t2507.name, t2510.name) AS "2_prov_2"
  FROM (SELECT 1 AS "2") AS t2512, departments AS t2507, employees AS t2510
 WHERE (t2507.name = t2510.dept AND t2510.salary > 1000000) OR t2510.salary < 1000;

(SELECT 3 AS "1_1", t2514."2" AS "1_2", t2514."1_1_oid" AS "2_row",
        'departments' AS "2_table"
      FROM (SELECT t2507.name AS "1_1_name", t2507.oid AS "1_1_oid",
                   t2510.dept AS "1_2_dept", t2510.name AS "1_2_name",
                   t2510.oid AS "1_2_oid", t2510.salary AS "1_2_salary",
                   row_number() OVER (ORDER BY t2507.name, t2510.name) AS "2"
              FROM departments AS t2507, employees AS t2510
             WHERE    (t2507.name = t2510.dept AND t2510.salary > (1000000))
                   OR t2510.salary < (1000)) AS t2514)
 UNION ALL
(SELECT 3 AS "1_1", t2516."2" AS "1_2",
        t2516."1_2_oid" AS "2_row", 'employees' AS "2_table"
   FROM (SELECT t2507.name AS "1_1_name", t2507.oid AS "1_1_oid",
                t2510.dept AS "1_2_dept", t2510.name AS "1_2_name",
                t2510.oid AS "1_2_oid", t2510.salary AS "1_2_salary",
                row_number() OVER (
                 ORDER BY t2507.name, t2510.name ) AS "2"
           FROM departments AS t2507, employees AS t2510
          WHERE (t2507.name = t2510.dept
                 AND t2510.salary > (1000000)
                ) OR t2510.salary < (1000) ) AS t2516 );
```

Figure 5.19: Generated lineage query Q7.

```
SELECT 0 AS "1_1", t2572."2" AS "1_2", t2543.name AS "2_data_a",
 t2560.name AS "2_data_b", 7 AS "2_data_c_1", row_number() OVER ( ORDER BY
 t2572."2", t2543.name, t2560.name ) AS "2_data_c_2", 7 AS "2_prov_1",
 row_number() OVER (ORDER BY t2572."2", t2543.name, t2560.name) AS "2_prov_2"
  FROM (SELECT 1 AS "2") AS t2572, employees AS t2543, employees AS t2560
 WHERE t2543.dept = t2560.dept AND t2543.name != t2560.name;

(SELECT 7 AS "1_1", t2574."2" AS "1_2", 'a' AS "2_data_doer",
        t2567.task AS "2_data_task", 2 AS "2_prov_1",
        row_number() OVER (ORDER BY t2574."2", t2567.oid) AS "2_prov_2"
   FROM (SELECT t2543.dept AS "1_1_dept", t2543.name AS "1_1_name",
                t2543.oid AS "1_1_oid", t2543.salary AS "1_1_salary",
                t2560.dept AS "1_2_dept", t2560.name AS "1_2_name",
                t2560.oid AS "1_2_oid", t2560.salary AS "1_2_salary",
                row_number() OVER (ORDER BY t2543.name, t2560.name) AS "2"
           FROM employees AS t2543, employees AS t2560
          WHERE t2543.dept = t2560.dept AND t2543.name != t2560.name
        ) AS t2574, tasks AS t2567
  WHERE t2574."1_1_name" = t2567.employee) UNION ALL
(SELECT 7 AS "1_1", t2576."2" AS "1_2", 'b' AS "2_data_doer",
        t2568.task AS "2_data_task", 4 AS "2_prov_1",
        row_number() OVER ( ORDER BY t2576."2", t2568.oid
        ) AS "2_prov_2"
   FROM (SELECT t2543.dept AS "1_1_dept", t2543.name AS "1_1_name",
                t2543.oid AS "1_1_oid", t2543.salary AS "1_1_salary",
                t2560.dept AS "1_2_dept", t2560.name AS "1_2_name",
                t2560.oid AS "1_2_oid", t2560.salary AS "1_2_salary",
                row_number() OVER ( ORDER BY t2543.name, t2560.name
                ) AS "2"
           FROM employees AS t2543, employees AS t2560
          WHERE t2543.dept = t2560.dept AND t2543.name != t2560.name
        ) AS t2576, tasks AS t2568
  WHERE t2576."1_2_name" = t2568.employee);
```

Figure 5.20: Generated lineage query QC4 (part 1).

```
(SELECT 2 AS "1_1", t2578."2" AS "1_2", t2578."1_3_oid" AS "2_row", 'tasks' AS "2_table"
   FROM (SELECT t2543.dept AS "1_1_dept", t2543.name AS "1_1_name",
                t2543.oid AS "1_1_oid", t2543.salary AS "1_1_salary",
                t2560.dept AS "1_2_dept", t2560.name AS "1_2_name",
                t2560.oid AS "1_2_oid", t2560.salary AS "1_2_salary",
                t2567.employee AS "1_3_employee", t2567.oid AS "1_3_oid",
                t2567.task AS "1_3_task", row_number() OVER (
                 ORDER BY t2543.name, t2560.name, t2567.oid ) AS "2"
           FROM employees AS t2543, employees AS t2560, tasks AS t2567
          WHERE     (t2543.dept = t2560.dept AND t2543.name != t2560.name)
                AND t2543.name = t2567.employee ) AS t2578) UNION ALL
(SELECT 4 AS "1_1", t2580."2" AS "1_2", t2580."1_3_oid" AS "2_row", 'tasks' AS "2_table"
   FROM (SELECT t2543.dept AS "1_1_dept", t2543.name AS "1_1_name",
                t2543.oid AS "1_1_oid", t2543.salary AS "1_1_salary",
                t2560.dept AS "1_2_dept", t2560.name AS "1_2_name",
                t2560.oid AS "1_2_oid", t2560.salary AS "1_2_salary",
                t2568.employee AS "1_3_employee", t2568.oid AS "1_3_oid",
                t2568.task AS "1_3_task", row_number() OVER (
                 ORDER BY t2543.name, t2560.name, t2568.oid ) AS "2"
           FROM employees AS t2543, employees AS t2560, tasks AS t2568
          WHERE (t2543.dept = t2560.dept AND t2543.name != t2560.name)
                AND t2560.name = t2568.employee ) AS t2580);

(SELECT 7 AS "1_1", t2582."2" AS "1_2", t2582."1_1_oid" AS "2_row", 'employees' AS "2_table"
   FROM (SELECT t2543.dept AS "1_1_dept", t2543.name AS "1_1_name",
                t2543.oid AS "1_1_oid", t2543.salary AS "1_1_salary",
                t2560.dept AS "1_2_dept", t2560.name AS "1_2_name",
                t2560.oid AS "1_2_oid", t2560.salary AS "1_2_salary",
                row_number() OVER (ORDER BY t2543.name, t2560.name) AS "2"
           FROM employees AS t2543, employees AS t2560
          WHERE t2543.dept = t2560.dept AND t2543.name != t2560.name
       ) AS t2582  ) UNION ALL
(SELECT 7 AS "1_1", t2584."2" AS "1_2", t2584."1_2_oid" AS "2_row", 'employees' AS "2_table"
   FROM (SELECT t2543.dept AS "1_1_dept", t2543.name AS "1_1_name",
                t2543.oid AS "1_1_oid", t2543.salary AS "1_1_salary",
                t2560.dept AS "1_2_dept", t2560.name AS "1_2_name",
                t2560.oid AS "1_2_oid", t2560.salary AS "1_2_salary",
                row_number() OVER ( ORDER BY t2543.name, t2560.name ) AS "2"
           FROM employees AS t2543, employees AS t2560
          WHERE t2543.dept = t2560.dept AND t2543.name != t2560.name) AS t2584);
```

Figure 5.21: Generated lineage query QC4 (part 2).

```
SELECT 0 AS "1_1", t2523."2" AS "1_2", t2518.name AS "2_data_1",
       t2521.name AS "2_data_2", 3 AS "2_prov_1", row_number()
       OVER (ORDER BY t2523."2", t2518.name, t2521.name) AS "2_prov_2"
  FROM (SELECT 1 AS "2") AS t2523, employees AS t2518, employees AS t2521
 WHERE (t2518.dept = t2521.dept AND t2518.salary = t2521.salary)
       AND t2518.name != t2521.name;

(SELECT 3 AS "1_1", t2525."2" AS "1_2", t2525."1_1_oid" AS "2_row",
        'employees' AS "2_table"
   FROM (SELECT t2518.dept AS "1_1_dept", t2518.name AS "1_1_name",
                t2518.oid AS "1_1_oid", t2518.salary AS "1_1_salary",
                t2521.dept AS "1_2_dept", t2521.name AS "1_2_name",
                t2521.oid AS "1_2_oid", t2521.salary AS "1_2_salary",
                row_number() OVER (ORDER BY t2518.name, t2521.name) AS "2"
           FROM employees AS t2518, employees AS t2521
          WHERE (t2518.dept = t2521.dept AND t2518.salary = t2521.salary)
            AND t2518.name != t2521.name ) AS t2525) UNION ALL
(SELECT 3 AS "1_1", t2527."2" AS "1_2",
        t2527."1_2_oid" AS "2_row", 'employees' AS "2_table"
   FROM (SELECT t2518.dept AS "1_1_dept", t2518.name AS "1_1_name",
                t2518.oid AS "1_1_oid", t2518.salary AS "1_1_salary",
                t2521.dept AS "1_2_dept", t2521.name AS "1_2_name",
                t2521.oid AS "1_2_oid", t2521.salary AS "1_2_salary",
                row_number() OVER (ORDER BY t2518.name, t2521.name) AS "2"
           FROM employees AS t2518, employees AS t2521
          WHERE (t2518.dept = t2521.dept AND t2518.salary = t2521.salary)
            AND t2518.name != t2521.name) AS t2527);
```

Figure 5.22: Generated lineage query QF3.

```
(SELECT 0 AS "1_1", t2535."2" AS "1_2",
        t2532.employee AS "2_data", 2 AS "2_prov_1",
        row_number() OVER (ORDER BY t2535."2", t2532.oid) AS "2_prov_2"
   FROM (SELECT 1 AS "2") AS t2535, tasks AS t2532
  WHERE t2532.task = ('abstract'))
UNION ALL
(SELECT 0 AS "1_1",
        t2537."2" AS "1_2",
        t2533.name AS "2_data",
        4 AS "2_prov_1",
        row_number() OVER (
         ORDER BY t2537."2", t2533.name
        ) AS "2_prov_2"
   FROM (SELECT 1 AS "2") AS t2537, employees AS t2533
  WHERE t2533.salary > (50000));

(SELECT 2 AS "1_1", t2539."2" AS "1_2",
        t2539."1_1_oid" AS "2_row", 'tasks' AS "2_table"
   FROM (SELECT t2532.employee AS "1_1_employee", t2532.oid AS "1_1_oid",
                t2532.task AS "1_1_task",
                row_number() OVER (ORDER BY t2532.oid) AS "2"
           FROM tasks AS t2532
          WHERE t2532.task = ('abstract')) AS t2539)
UNION ALL
(SELECT 4 AS "1_1", t2541."2" AS "1_2",
        t2541."1_1_oid" AS "2_row", 'employees' AS "2_table"
   FROM (SELECT t2533.dept AS "1_1_dept", t2533.name AS "1_1_name",
                t2533.oid AS "1_1_oid", t2533.salary AS "1_1_salary",
                row_number() OVER (ORDER BY t2533.name) AS "2"
           FROM employees AS t2533
          WHERE t2533.salary > (50000) ) AS t2541 );
```

Figure 5.23: Generated lineage query QF4.

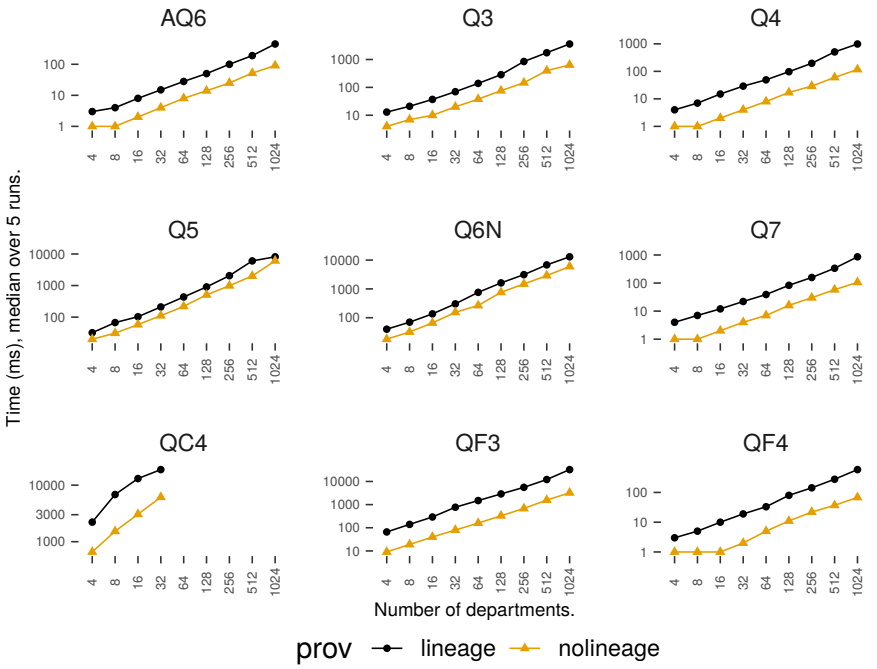

Figure 5.24: Lineage query runtimes.

Query	#depts	median runtime in ms		overall slowdown
		lineage	nolineage	(geom mean)
AQ6	1024	450	91	3.79
Q3	1024	3597	637	3.98
Q4	1024	986	117	6.66
Q5	1024	8149	6093	1.91
Q6N	1024	13007	5979	2.21
Q7	1024	861	106	5.73
QC4	32	18841	6128	1.80
QF3	1024	32498	3240	8.36
QF4	1024	584	68	7.45

Figure 5.25: Median runtimes at largest data set and geometric means of overall slowdowns across all instance sizes.

queries QF3 and QF4 are among the worst, while having some of the largest and smallest execution times overall.

i	s	cardinal	ordinal
1	"1"	"one"	"first"
2	"2"	"two"	"second"
⋮			
n	"*n*"	"*en*"	"*n*th"

Figure 5.26: Synthetic taste of 64 tables with $n = 10^4, 10^5, 10^6$ rows each.

5.3 Comparison with Perm

In this section we compare Links^W and Links^L to Perm [Glavic and Alonso, 2009], as one instance of a database-integrated provenance system. This is very much a comparison between apples and oranges.

The subset of queries supported by both Links variants and Perm is limited. Most of the queries above use nested results which are not supported by Perm. Many common flat relational queries use aggregations which are not supported by Links. Others do not have large or interesting provenance annotations, be it where-provenance or lineage.

For this comparison we use a synthetic test database as illustrated in Figure 5.26. We create tables of integers $1, \ldots, n$ for $n = (10000, 100000, 1000000)$; a simple string representation of the number; an English language cardinal ("one", "two", ...); and an English language ordinal ("first", "second", ...). We create 64 copies of these tables at each size n and call them i_s_c_o_n_1, i_s_c_o_n_2, ... Their content is the same, but their `oid`s are different. The data loading scripts are 55 MB, 640 MB, and 7.8 GB on disk.

We use the same machine as before to run both databases and database clients. We used Perm version 0.1.1 which is a fork of PostgreSQL 8.3. There is a known problem compiling PostgreSQL 8.3 with a current Gcc (6.3.1) which requires passing `-fno-aggressive-loop-optimizations`. This has been fixed in later versions of PostgreSQL. One of the advantages of language-integrated provenance is that it works independently of the database. Links uses the current version of PostgreSQL as its database back-end, which at the time these benchmarks were run was PostgreSQL 9.6.3.

We measure wall clock time of single runs of a complete program. Links executes the query and prints the result to stdout which is ignored. Printing uses

Links's native format with pretty printing (colors, line breaks, and indentation) disabled. Perm queries are executed using Psql with a "harness" like this to produce output in comma-separated value format:

```
\COPY (SQL query goes here) TO STDOUT WITH CSV
```

5.3.1 Links^W

We use a family of queries that join $m = (16, 32, 64)$ of the tables described above on their integer column and select the provenance-annotated cardinal column for each of them. Thus, the where-provenance Links^W queries look like this:

```
for (t_1 <-- i_s_c_o_n_1) ... for (t_m <-- i_s_c_o_n_m)
where (mod(t_1.i,100) < 5 && t_1.i==t_2.i && ... && t_1.i==t_m.i)
  [(c1=t_1.cardinal, c2=t_2.cardinal, ..., cm=t_m.cardinal)]
```

Testing revealed that Links^W runs out of memory for the largest (n=1000000, m=64) query when loading the results. Rather than using smaller input databases, we filtered the result using `mod(t_1.i,100) < 5` as an additional condition in the where clause.

Unfortunately, *Perm's* where-provenance support is too restrictive and refuses to execute an equivalent query with the following error message: "WHERE-CS only supports conjunctive equality comparisons in WHERE clause." Fortunately, Perm has no problems computing the full result, so we used queries of the following form, *without* filtering based on `t_1.i % 100 < 5`.

```
SELECT
  PROVENANCE ON CONTRIBUTION (WHERE)
    t_1.cardinal AS c1, ..., t_m.cardinal AS cm
FROM i_s_c_o_n_1 AS t_1, ..., i_s_c_o_n_m AS t_m
WHERE t_1.i = t_2.i AND ... AND t_1.i = t_m.i
```

We use variants without where-provenance of both the Links^W and Perm queries. For the provenance-less Links^W version, we keep the table declarations as they are, but use the **`data`** keyword to project to just the data and rely on query normalization to not compute provenance. For the provenance-less Perm version, we use a plain `SELECT` without the `PROVENANCE` clause.

Finally, we have a fifth set of queries that is just like the plain Perm queries, but *with* filtering. We run these against PostgreSQL 9.6.3, the version Links^W uses. This gives us an idea of the overhead for loading and printing incurred by Links compared to Psql.

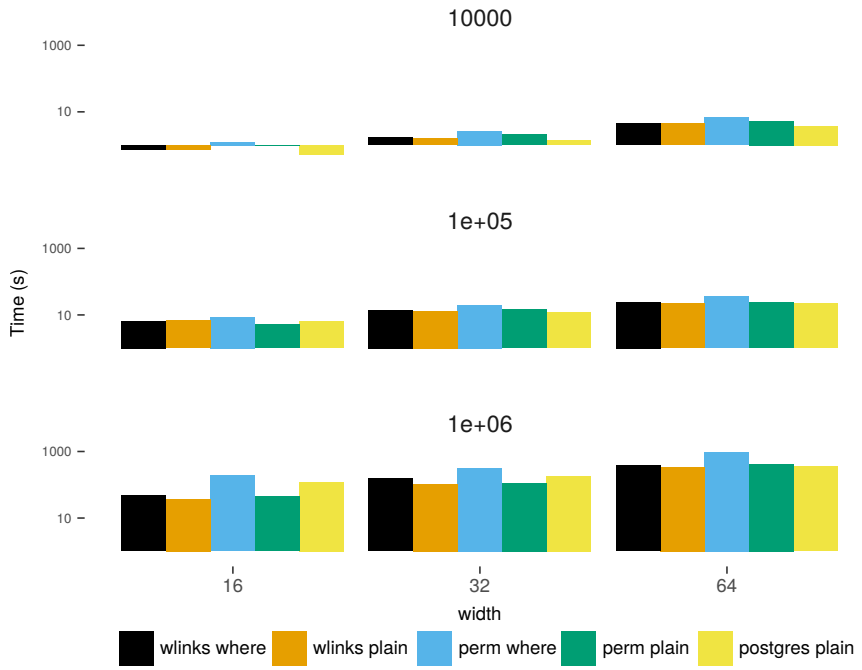

Figure 5.27: Where-provenance times grouped by table size (n) and number of tables (m). Note that wlinks and postgres queries are filtered, perm queries are not.

The Links^W results with where-provenance enabled look something like the following with pretty printing of provenance-annotated values disabled:

```
[(c1=(!data="one",!prov=("i_s_c_o_10000_1", "cardinal", 715924950)),
  c2=(!data="one",!prov=("i_s_c_o_10000_2", "cardinal", 715925958))...)...]
```

Perm uses arrays to collect annotations of equal rows. In our query, all rows are different, so these are all singleton arrays.

c1	annot_c1	…
two hundred sixty-seven	{public.i_s_c_o_10000_1#cardinal#114040340}	…
three hundred seventeen	{public.i_s_c_o_10000_1#cardinal#114040390}	…
⋮	⋮	

Figure 5.27 shows query runtimes in seconds grouped by size of tables (n) and number of tables joined (m). Keep in mind that the Perm variants are not filtered. Figure 5.28 shows the result size in megabytes at $n = 1000000$ for Links^W and Perm with where-provenance annotations, and PostgreSQL without annotations. We measure the size simply as byte count of the printed result.

	m=16	m=32	m=64
Links^{W}	89.2 MB	179.1 MB	359.1 MB
Perm	1589.3 MB	3187.5 MB	6384.0 MB
PostgreSQL	37.2 MB	74.3 MB	148.6 MB

Figure 5.28: Result sizes at n=1000000. Links^{W} and Perm results are with where-provenance annotations, PostgreSQL is without.

Looking at the runtime difference between the Perm queries without where-provenance and the plain PostgreSQL queries we see that the result size does not have a great impact on runtime. In general, the numbers between systems are hard to compare, not just because of result size. We only consider one family of highly synthetic queries and the experimental setup is not necessarily a realistic reflection of any real-world use. However, we do observe some trends: The runtime difference between processing 10x data (going down one row in the graph) is larger than the difference between systems, by far. Doubling the number of tables considered also dominates difference between systems. We conclude that the overhead of where-provenance in both Perm and Links^{W} is moderate and the systems are roughly comparable.

5.3.2 Links^{L}

We use the same data as before and similar queries to compare Links^{L} to *Perm Influence Contribution Semantics* (PI-CS). Lineage and PI-CS are not equivalent in general [Glavic, 2010], but for the queries we use here the annotations contain, more or less, the same information.

We use a family of queries similar to those for where-provenance. Again we join $m = (16, 32, 64)$ tables, but this time we return only the first table's integer and English cardinal columns, and their lineage. The number of joins is particularly interesting here because it increases the size of the provenance metadata without affecting the actual result size.

The Links^{L} lineage versions of the query look like the following:

```
lineage {
  for (t_1 <-- i_s_c_o_n_1) ... for (t_m <-- i_s_c_o_n_m)
  where (mod(t_1.i,100)<5 && t_1.i==t_2.i&&...&&t_(m-1).i==t_m.i)
    [(i = t_1.i, c = t_1.cardinal)] }
```

The plain, unannotated versions just use **query** in place of **lineage**.

The Perm versions look like the one below, with and without PROVENANCE. Note that in this set of benchmarks the Perm queries are also filtered to 5% of the result.

```
SELECT PROVENANCE t_1.i, t_1.cardinal
FROM i_s_c_o_n_1 AS t_1, ..., i_s_c_o_n_m AS t_m
WHERE t_1.i%100<5 AND t_1.i=t_2.i AND ... AND t_(m-1).i=t_m.i
```

Instead of a list of annotations per result row, Perm produces wider tables, adding columns to identify join partners. Table rows are identified by their whole contents, so for $m = 64$ joined tables we have two columns for the actual result and $64 * 4$ columns of provenance metadata. The example result below is transposed and shows the first two rows and first eight columns of the result.

i	1	2	...
cardinal	one	two	...
prov_public_i_s_c_o_1000_1_i	1	2	...
prov_public_i_s_c_o_1000_1_s	1	2	...
prov_public_i_s_c_o_1000_1_cardinal	one	two	...
prov_public_i_s_c_o_1000_1_ordinal	first	second	...
prov_public_i_s_c_o_1000_2_i	1	2	...
prov_public_i_s_c_o_1000_2_s	1	2	...
⋮	⋮	⋮	

We show query runtimes grouped by size of the tables (n) and number of tables joined (m) in Figure 5.29. We omitted the largest Links$^{\mathrm{L}}$ query (n=1000000, m=64); it ran for 33745 seconds, which would have distorted the graph too much. This query just barely did not run out of memory, causing severe GC thrashing and leaving little memory for the database server and disk caches.

These timings are whole program execution and so include pre- and post-processing steps. Links$^{\mathrm{L}}$ is an extension of Links with minimal changes to the parser and type checker that desugars an early syntax tree to plain Links more or less as described in Section 4.3. This takes less than 1 millisecond for all queries. Query normalization for the lineage queries takes around 9 milliseconds for m=16, 41 milliseconds for m=32, and 194 milliseconds for m=64. Post-processing times (with data already in memory) range from 11

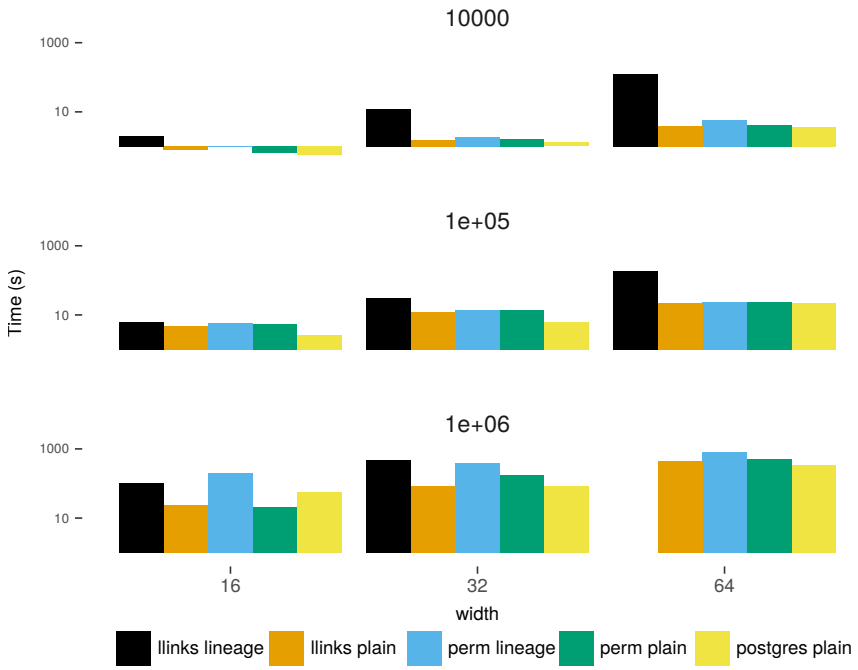

Figure 5.29: Lineage times grouped by relation size (n) and width (m). All queries are filtered to return only 5% of results.

milliseconds for n=10000, m=16 to almost 10 seconds for the lineage query at n=1000000, m=64.

The queries executed by PostgreSQL are on average a bit faster than the same queries executed by Perm. We did not investigate this further, a simple explanation would be that PostgreSQL 9.6.3 is just that bit faster than PostgreSQL 8.3 which is the version Perm was forked from. Because Links works with any recent PostgreSQL version, it is easy to take advantage of such performance improvements in the database.

Figure 5.30 shows the result size at $n = 1000000$ for plain queries, and lineage queries at $m = 16$ and $m = 32$. In some ways, the data is a worst case for Perm, because the width of the result is so much smaller than the width of the annotations. Despite that, the query execution time overhead of lineage annotations is remarkably low in Perm.

While Perm considerably outperforms Links^{L} on lineage computations, their performance on plain queries is similar, which comes as a bit of a surprise. We expected Links^{L} to be a worse database client than the native Psql client, even for flat queries. In part this is due to the experimental setup with both database and client running on the same machine. Because Links^{L} uses so much memory,

	plain query	lineage (m=16)	lineage (m=32)
Links^{L}	3.1 MB	38.5 MB	73.7 MB
Perm	2.7 MB	89.4 MB	176.2 MB

Figure 5.30: Size of printed results at n=1000000.

there is less memory available for disk caches and the database system spends a lot of time waiting for disk seeks. Post-processing time is low by comparison: except for the largest queries, post-processing by Links^{L} is typically well below one second.

5.4 Discussion

We discuss threats to the validity of our results and conclusions, aspects of provenance we did not benchmark but would like to see benchmarked in the future, benchmarks by Lee et al. [2018], and conclude that the performance of language-integrated provenance is reasonable.

The organization database that is used in the first two sections is quite small, even at the largest size with 4096 departments. We have addressed this concern in part in the comparison with Perm, which uses a much larger database that does not easily fit into memory.

The results on the other hand are perhaps unrealistically large. Query Q1, for example, returns the whole database in a different shape. One of the envisioned main use cases of provenance is debugging. Typically, a user would filter a query to focus on a surprising result and thus query less provenance. Our experiments do not measure this scenario but instead compute provenance eagerly for all query results. Thus, the slowdown factors we obtain represent a worst case upper bound that may not be experienced in common usage patterns.

One problem with large results is that Links' runtime representation of values has a large memory overhead. This is particularly problematic in our benchmark setup because we run database and client on the same machine where they compete for memory. In practice, for large databases we should avoid holding the whole result in memory. (It is not entirely clear how to do this in the presence of nested results and thus query shredding.) It could also

be advantageous to represent provenance in a special way. There is typically a lot of repetition in relation and column names for example.

It would be interesting to run experiments where database and client run on different machines. Links^W and Links^L work with any PostgreSQL database[2] and it would be particularly interesting to test them on a hosted service like Amazon Rds that is tuned for performance by professionals. We are not aware of any Perm-as-a-service offerings, but there are other provenance systems that work with PostgreSQL (see Section 2.1.2).

The measurements in the first two sections do not include program rewriting time. However, this time is only dependent on the lexical size of the program and is thus fairly small and, most importantly, independent of the database size. Since Links is interpreted, it does not really make sense to distinguish translation time from execution time, but both the where-provenance translation and the lineage translation could happen at compile time, leaving only slightly larger expressions to be normalized at runtime. Across the queries in Sections 5.1 and 5.2, the largest observed time spent rewriting Links^W or Links^L to plain Links was 5 milliseconds with an average of 0.5 milliseconds.

Our performance evaluation does not include any queries on provenance data itself. All three of Links^W, Links^L, and Perm use a flat relational representation of provenance[3] and offload computation on provenance to the database engine, just like computation of the query data itself. Flat relations are the bread and butter of database systems and so we expect them to do a good job at planning efficient queries on provenance. It would be interesting to test this hypothesis by comparing language-integrated provenance against systems that either cannot filter on provenance computations in the same query at all, or make heavy use of procedural features and user-defined functions which are challenging for query optimizers to see through. Müller et al. [2018] focus on preserving the shape of queries when computing provenance and explicitly reject propagating provenance together with data through the query which would make filtering based on provenance easy. Compared to Perm they observe drastic speedups for some queries and drastic slowdowns for others. Unfortunately, they did not compare any queries that perform computation based on provenance —

[2]Query shredding relies on the existence of a database driver for Links (currently PostgreSQL, MySQL, and SQLite are known to work) and database support for `ROW_NUMBER`.

[3]via shredding, in the case of Links^L, but flat nonetheless

exactly the case where we would expect the likes of Links^{W}, Links^{L}, and Perm to perform particularly well.

The prototypes of Links^{W} and Links^{L} we benchmarked in this chapter are based on Links and therefore on query shredding [Cheney et al., 2014c]. Stolarek and Cheney [2018] implemented language-integrated provenance in the spirit of Links^{W} and Links^{L} on top of the Haskell library Dsh [Ulrich and Grust, 2015]. Dsh uses a compilation strategy based on the flattening transformation [Blelloch and Sabot, 1990] and claims to be an improvement over the previous implementation based on loop-lifting [Giorgidze et al., 2011; Grust et al., 2010]. Stolarek and Cheney [2018] did not perform any benchmarks and there is no direct comparison between query shredding and flattening. Nevertheless, we would not be surprised if language-integrated query in Haskell based on Dsh was faster than the Links^{W} and Links^{L} prototypes. If nothing else it would still have a more compact in-memory representation of results.

Lee et al. [2018] compared Pug against Links^{L} using queries Q7 and QF3 from Section 5.2. They compared Pug to three versions of the Links^{L} queries: (1) the whole Links^{L} program, including post processing the shredded query and loading data into the in-memory representation; (2) just the Sql queries generated by Links^{L}; and (3) the Sql queries generated by Links^{L} plus additional joins to fetch the whole contents of the row represented by a lineage annotation, not just table and row number. For small database instances, Links^{L} in all variants performs better than Pug. Starting at 1024 departments, Links^{L} variant 1 is slower; variant 2 stays faster; and variant 3 is even. In general they observe much larger Links^{L} post processing overheads than we did. This is likely due to better experimental setup on a machine with 16 times more memory, which protects the database system and its disk caches from Links^{L}'s excessive memory use. For small databases, variant 2 is more than twice as fast as Pug, and at 2048 departments (the largest instance they tested) it is still 1.5 times faster, which suggests that with a more careful implementation, language-integrated provenance could indeed stand its ground. Pug returns the whole row as lineage, not just its table and row number like Links^{L} does. Benchmark variant 3 is an attempt to make the results of Links^{L} and Pug more comparable. However, we believe that this comparison is not quite accurate. Because annotations can come from multiple tables, there must be as many additional joins as there are source tables in a query. If Links^{L} natively used whole rows as provenance, it would

propagate them through the query without additional joins; however, it is not clear how to do this in a well-typed way without resorting to dependent types to allow the result type to depend on the query structure.

The comparison of Links^{L} and Perm suggest that perhaps we should look into generating more efficient queries. Currently, we use a naive translation from Links^{L} to Links which relies on Links's general purpose nested query capabilities. Perm exploits the fact that lineage is bounded by the structure of the query, adding just the right number of columns instead of arbitrarily large nested data. Exposing this to the programmer would, again, require fancy types. However, perhaps Links^{L} could do this internally only, post-process the result, and return nested collections to the programmer. Similarly, should database-integrated provenance, Perm or otherwise, ever become too fast to compete with and be in widespread use, we can always generate queries that use the provenance features of the underlying database, all without leaving the comfort of writing composable queries in a type-safe programming language.

Despite all the difficulties in benchmarking, I think we can conclude that language-integrated provenance is at least not hopelessly inefficient. Compared to plain queries, the overhead of where-provenance in Links^{W} ranges from 1.3 to 2.8 and the overhead of lineage in Links^{L} ranges from 1.8 to 8.4. Bear in mind that it is not at all obvious that manually written queries which compute the same information would be any faster. Compared to Perm, a database-integrated provenance system, overheads in Links^{W} and Links^{L} are bigger, but mostly not orders of magnitude bigger. Compared to Pug, which is a middle-ware–type system, even the naive implementation of Links^{L} performs quite well.

Chapter 6

LINKS^{T} — provenance through trace analysis

A paper [Fehrenbach and Cheney, 2019] based on the main ideas in this chapter has been accepted for publication.

Chapters 3 and 4 have shown how one could go about adding support for where-provenance and lineage, respectively, to a programming language. This chapter explores what features a programming language requires to support multiple forms of provenance in the same language, possibly even in the same query, and how programmers could define their own forms of provenance.

The high-level idea is as follows: We change database queries to compute a trace of their execution instead of their result. Programmers can then use ordinary functions[1] to inspect query traces and extract information such as where-provenance or lineage. As before, we use query normalization to avoid actually constructing any intermediate trace information if the result of composing a trace analysis function with a query trace has a nested relational type.

The challenge is to find a sweet spot where traces contain enough information to be useful, the language is powerful enough to write generic, well-typed trace analysis functions, and at the same time simple enough to still normalize to efficient queries.

The next section gives a high-level overview of LINKS^{T} and introduces some of its features by example. Section 6.2 describes the syntax and semantics in detail. Section 6.3 uses LINKS^{T} to define some trace analysis functions, including where-provenance and lineage. Section 6.4 discusses how to turn query expressions

[1] We will need to add a small number of generic programming features to the language.

into expressions that produce a trace of their own execution. Section 6.5 extends query normalization to deal with the new LINKS^T features. We have implemented a prototype of the self-tracing transformation and normalization in HASKELL and discuss its output on some example queries in Section 6.6. We discuss related and future work in Section 6.7.

6.1 Overview

The implementations of LINKS^W and LINKS^L are separate extensions of LINKS. This does not have a deep technical reason; they could coexist in a single language. Indeed, Stolarek and Cheney [2018] have shown how to implement both where-provenance and lineage in HASKELL using similar query transformations. It is even conceivable to apply both transformations to the same query.[2] We thus claim that language-integrated provenance satisfies Requirement 1: "support for different types of provenance" [Glavic et al., 2013]. However, the situation is not ideal. Extending LINKS in the manner of LINKS^W and LINKS^L requires changes to the parser, typechecker, and interpreter. This is not something regular programmers should be expected to do. The situation in HASKELL is not much better. While Stolarek and Cheney [2018] did not need to change the implementation of HASKELL, they did have to change the implementation of DSH [Giorgidze et al., 2011; Ulrich and Grust, 2015] to, among other things, "implement our own type checking for some fragments of the lineage transformation". We would much prefer programmers be able to define their own forms of provenance without changing the implementation of their query language (whether it is embedded in another language or not).

LINKS^T takes inspiration from work on *slicing* database queries [Cheney et al., 2014a] and (imperative) functional programs [Perera et al., 2012; Ricciotti et al., 2017]. The common theme in this line of work is tracing execution to have enough information to answer questions about which parts of the program are responsible for which parts of the output. They define augmented evaluation procedures that not only produce a value, but also a trace of the execution and then further analyze the trace to obtain slicing information.

Recent work by Müller et al. [2018] is closely related to LINKS^T and in some

[2] Where-provenance first, then lineage, otherwise the former would try to annotate annotations from the latter. The latter applies only to list constructors, which the former ignores.

ways more advanced. The standard semantics of SQL queries is to produce a value when run on a database. Müller et al. [2018] show different interpretations of SQL queries that produce where-provenance and lineage of a query. Unlike in LINKS$^{\mathrm{T}}$ and in the slicing work, there is no single tree structured trace. They essentially decompose the single trace into two parts: a dynamic part which records control flow decisions the database system needs to make depending on the data; and a static part that is just the structure of the query. Their work also extends to SQL features like grouping and aggregation that are not implemented in LINKS, let alone traced in LINKS$^{\mathrm{T}}$.

In both the slicing work and the database-integrated provenance work, queries do not have access to the trace and it is not the query language that is used to interpret the traces. Instead, both the trace and its analysis live outside of the program. In contrast, LINKS$^{\mathrm{T}}$ traces are a recursive datatype in the language itself. We can wrap a piece of code in a **`trace`** block and get back a value — the trace that represents the code's execution. We can write (recursive) functions that analyze traces and extract information, like from any other datatype. We can even combine multiple different trace analyses.

The key constraint is designing the traces and trace analysis functions such that when we compose analysis and tracing of some code and the result has a query type, we can compile the code to a bounded number of SQL queries. That is, assuming normalization terminates, which we do not prove. Queries can be constructed at runtime to be arbitrarily large. To deal with arbitrarily large traces, we need to write recursive trace analysis functions. Recursive functions do not in general normalize — they may loop. We currently do not enforce termination through means like checking syntactically for structural recursion.

The rest of this section gives a high-level overview of LINKS$^{\mathrm{T}}$'s features. We describe type-level computation, how to make term-level decisions based on types, generic operations on records, and what traces look like by example.

6.1.1 Type functions and `Typerec`

Sometimes, it is useful to make decisions based on types. LINKS$^{\mathrm{T}}$ has two language features to support this: **`Typerec`** on the type-level and **`typecase`** on the term-level. Recall the lineage translation on types $\mathfrak{L}[\![\cdot]\!]$ (Figure 4.2 on page 59) that we used to define what lineage-annotated types look like: base types are

translated to themselves, for list types we recursively translate the type argument and add the actual lineage annotations, and records are just recursively translated. For LINKS^L this was implemented on the meta-level in OCAML. In LINKS^T we can use **Typerec** to define such a traversal:

```
Typerec a (Bool, Int, String,
           λb b'. [⟨data:b', lineage:[⟨table:String, row:Int⟩]⟩],
           λr r'. ⟨r'⟩)
```

You can read the snippet above as a fold over the type `a`. If `a` is one of `Bool`, `Int`, or `String` we return the same. On the second line we have a type function of two arguments that defines what to do if `a` is a list type. It is called with the first argument `b` bound to the element type of `a` and the second argument `b'` bound to the result of applying the **Typerec** expression recursively to the element type of `a`. The result in our case is a list type where the elements are records containing recursively traversed lineage-annotated types and a list of lineage annotations. The third line defines another type-level function. It gets called when `a` is a record type and gets passed the original row, as well as the row obtained by recursively evaluating the **Typerec** expression on all field types. In our case we want to recursively go through records, so we return a record with row `r'`.

6.1.2 Type-directed programming with `typecase`

On the term-level we have **typecase** to make decisions based on types. A typical application is overloading functions, for example for pretty printing. Perhaps the simplest use case is to define a default value at any type, as below.

```
default : ∀a:Type.a
default = Λa:Type.typecase a of
  Bool   => false
  Int    => 0
  String => ""
  List b => []
```

This snippet defines `default`, a polymorphic value that has any type in a universe of base types and list types. It is defined as a type abstraction Λ that binds type `a` and then pattern matches on the shape of `a` using **typecase**. The base type branches return some typical default value. The list branch binds element

type `b` and returns the empty list of type `List b`. Using this definition we have `default Int == 0` and `default (List (List Bool)) == []`.

In Links^T the universe of types that can be inspected by **`typecase`** additionally includes record types and trace types. Ad-hoc polymorphism is generally useful, but for us it is crucial for writing generic functions that can later be applied to analyze traces of queries of any type. This and the next feature are what allows us to write a trace analysis function once and then use the same function to compute provenance of queries with any type.

6.1.3 Generic record programming

Since we want to be able to write a single trace analysis function and apply it to a variety of queries, we need to be able to work generically with records. Links^T supports two generic operations on records: mapping and folding.

For example, we can convert a record to a record of strings, by mapping a polymorphic `show` function over it like so:

```
rmap show ⟨a=true, b=[false]⟩
```

with the following result: ⟨a="*true*", b="[*false*]"⟩.

The expression in function position, `show` in this case, must have the following polymorphic function type: $\forall\alpha.\alpha \to F\,\alpha$, where F is a type-level function. In our example, F is the constant type-level function $\lambda\alpha : \mathit{Type}.\texttt{String}$. For closed records, the result type of a record map expression is another record type with the type-level function F applied to the labels' types. For open records, that is records whose row contains a row variable ρ, we keep track of the fact that we apply a type-level function to the labels' types using the type-level row map operation **`Rmap`**. Given a type-level function F and a row type R, **`Rmap`** F R represents the row type that applies F to all of R's labels' types. For example, the following types are all equivalent:

```
  ⟨a: String, b: String⟩
= Record (a: String, b: String)
= Record (a: (λa.String) Bool, b: (λa.String) (List Bool))
= Record (Rmap (λa.String) (a: Bool, b: List Bool))
= (λr.Record (Rmap (λa.String) r)) (a: Bool, b: List Bool)
```

The second generic record operation is folding. We will need this to combine annotations from the fields of a record, for example in the lineage implementation in Section 6.3.3. For this introduction, let us assume we wanted to count the

number of times **true** appears in a record. Given a polymorphic `countTrue` function, we could map it over the record, and then use a record fold operation to further reduce the resulting homogeneous record into the final sum:

```
rfold (+) 0 (rmap countTrue ⟨a=true, b=[true, false], c=5⟩)
```

The result of mapping is ⟨a=1, b=1, c=0⟩ and folding with addition and initial accumulator 0 results in 2. As the order of labels in records is undefined, the order of traversal by **rmap** and **rfold** is undefined too. Since we only have pure functions in queries, the order of execution makes no difference to **rmap**. When using **rfold** however, it is best to use a commutative combining function.

During LINKS^T query compilation, **rmap** and **rfold** are unrolled based on types. This works because queries do not contain free (row) variables and all record types that appear in a query directly are closed (they are either literal record constructions, or come from tables which have fixed columns).

6.1.4 Traces of queries

What exactly do these traces look like? We answer this in detail in Section 6.4. For now, let us look at a few examples. The trace of the literal 42 is `Lit 42`. `Lit` is a constructor of the built-in type `Trace`. Tracing a list of literals like `[42, 43]` results in a list of traces like `[Lit 42, Lit 43]`. And tracing a record ⟨a: **true**, b: 42⟩ results in a record of traces ⟨a: Lit **true**, b: Lit 42⟩. This is perhaps the most important aspect of traces in LINKS^T: the trace of a list is a list of traces and, similarly, the trace of a record is a record of traces. Projection is not currently recorded in traces since we have not needed it, but it would be possible to add it. For now, ⟨a: **true**, b: 42⟩.a results in the trace `Lit` **true**. Tracing tables annotates all cells. For example, the trace of `presidents` from Figure 1.1 on page 2 would look something like this (modulo `oids`):

```
[⟨nth=Cell ⟨table="presidents", column="nth", row=432, data=1⟩,
  name=Cell ⟨table="presidents", column="name", row=432,
             data="George Washington"⟩⟩,
 ...,
 ⟨nth=Cell ⟨table="presidents", column="nth", row=564, data=44⟩,
  name=Cell ⟨table="presidents", column="name", row=564,
             data="Barack Obama"⟩⟩,
 ⟨nth=Cell ⟨table="presidents", column="nth", row=149, data=45⟩,
```

```
name=Cell ⟨table="presidents", column="name", row=149,
           data="Donald Trump"⟩⟩]
```

That is, data from table cells is annotated with the `Cell` constructor which carries table name, column name, and row identifier (we use `oid`s again) alongside the actual data. The other `Trace` constructors are: `Lit` for values from the query itself rather than the database; `If` for the result of an **`if`**–**`then`**–**`else`** annotated with subtraces of the conditional expression and the branch taken; `For` for values that were produced by a comprehension, recording a trace of the input and the output; and traces of primitive operators like == and + that record subtraces for their arguments.

Polymorphic operations record at which type they were applied. For example, the trace for `5 == 7` is `OpEq Int ⟨left=Lit 5, right=Lit 7⟩`. In addition to the left and right subtraces, we record that in this case equality was applied at type `Int`. Comprehensions are also polymorphic and record the type of the input collection's elements.

Figure 6.1 shows a partial trace of the larger (but really still quite small) boat tours query. It has almost the full trace for the name field of the first result row. We see how nested **`for`** comprehensions lead to nested `For` trace constructors, and nested comparisons in the conditional expression lead to nested `OpAnd` and `OpEq` traces. Note that this is for illustration purposes only. We do not intend to ever actually construct the whole trace. We expect to compose the self-tracing query with a trace analysis function and evaluate away all trace constructors during query normalization.

We use **`tracecase`** to pattern match, or rather case split, on trace constructors. The branches for `For` and `OpEq` bind an additional type variable. For example, the snippet below would evaluate to `f Int (Lit 5)`:

```
tracecase OpEq Int ⟨left=Lit 5, right=Lit 6⟩ of
  Lit x    => ...
  If x     => ...
  For a x  => ...
  Cell x   => ...
  OpEq a x => f a x.left
  OpPlus x => ...
```

This concludes the informal overview of Links$^{\mathsf{T}}$ features. The next sections describe the syntax and static semantics of Links$^{\mathsf{T}}$, including normalization to nested relational calculus, and the self-tracing transformation. If you are

```
[⟨name=
   For⟨in=⟨name=Cell⟨table="agencies", column="name",
                     row=?, data="EdinTours"⟩,
           phone=Cell⟨table="agencies", column="phone",
                     row=?, data="4121200"⟩,
           based_in=Cell⟨table="agencies", column="based_in",
                     row=?, data="Edinburgh"⟩⟩,
       out=
     For⟨in=⟨type=Cell⟨table="externalTours", column="type",
                       row=?, data="boat"⟩,
             name=...,
             price=...,
             destination=...⟩,
         out=
       If⟨cond=OpAnd⟨left=
                 OpEq String ⟨left=Cell⟨table="agencies",
                                        column="name",
                                        row=?,
                                        data="EdinTours"⟩,
                             right=Cell⟨table="externalTours",
                                        column="name",
                                        row=?, data="EdinTours"⟩⟩,
                     right=
                 OpEq String ⟨left=Cell⟨table="externalTours",
                                        column="type",
                                        row=?, data="boat"⟩,
                             right=Lit "boat"⟩⟩,
          branch=Cell⟨table="agencies", column="name",
                      row=?, data="EdinTours"⟩⟩⟩⟩
 phone=...⟩,
⟨name=..., phone=...⟩,
...]
```

Figure 6.1: Partial trace of the boat tours query.

inclined to skip the technical details for now, Section 6.3 shows Links^T in action. There we define trace analysis functions to extract provenance from traces.

6.2 Syntax & semantics

Links^T is meant to eventually be implemented as an extension to Links. However, the language design described here is more closely based on λ_i^{ML}, a core calculus devised by Harper and Morrisett [1995]. This presentation of the typing rules in particular is based on a later presentation of λ_i^{ML} by Crary et al. [2002]. We modify the core of λ_i^{ML} to accommodate Links features like row types, records, and lists. To simplify the presentation, Links^T is entirely pure but we imagine that an eventual implementation would use effect types much as Links does [Lindley and Cheney, 2012] to distinguish database-executable query expressions from other programs.

The syntax of Links^T is shown in Figure 6.2. We use a unified context Γ, mapping type variables x to types A and type variables α to kinds K. Kinds include *Type* and *BaseType*, as well as *Row* and *BaseRow*, a row of base types. Row constructors S include row variables ρ. We follow convention in using ρ for row variables, but they really are just type variables with kind *Row*.

Links^T, like λ_i^{ML}, has two syntactic categories for type-like things: types A and constructors C. Types are what we usually understand as types, they categorize values. Constructors are where type-level computation happens and they can be inspected by terms like **`typecase`**, whereas types cannot be inspected.

Types, unsurprisingly, contain base types, function types, list and record types. For the sake of brevity, we will often write `[`A`]` for `List` A and $\langle R \rangle$ for `Record` R. In rules, we will also use the shorthand syntax **`typecase`** C **`of`** $(M_B, M_I, M_S, \beta.M_L, \rho.M_R, \beta.M_T)$ for

```
typecase C of
  Bool      => M_B
  Int       => M_I
  String    => M_S
  List β    => M_L
  Record ρ  => M_R
  Trace β   => M_T
```

and similarly **`tracecase`** M **`of`** $(x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P)$ for

Contexts	Γ	::=	$\cdot \mid \Gamma, \alpha : K \mid \Gamma, x : A$
Kinds	K	::=	$\mathit{Type} \mid \mathit{BaseType} \mid \mathit{Row} \mid \mathit{BaseRow} \mid K_1 \to K_2$
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 constr.	S	::=	$\cdot \mid l^P : C; S \mid \rho \mid \textbf{Rmap}\ C\ S$
Types	A, B	::=	$\texttt{T}(C) \mid \texttt{Bool} \mid \texttt{Int} \mid \texttt{String} \mid A \to B$
		$\mid$	$\texttt{List}\ A \mid \texttt{Record}\ R \mid \texttt{Trace}\ A \mid \forall\alpha : K.A$
Row types	R	::=	$\cdot \mid l^P : A; R$
Presence types	P	::=	$\circ \mid \bullet$
Terms	L, M, N	::=	$c \mid x \mid \lambda x : A.M \mid M\ N \mid \Lambda\alpha : K.M \mid M\ C$
		$\mid$	$\textbf{fix}\ f : A.M \mid \textbf{if}\ L\ \textbf{then}\ M\ \textbf{else}\ N \mid M + N \mid M == N$
		$\mid$	$\langle\rangle \mid \langle l = M; N\rangle \mid M.l \mid \textbf{rmap}^S\ L\ M \mid \textbf{rfold}^S\ L\ M\ N$
		$\mid$	$[] \mid [M] \mid M +\!\!+ N \mid \textbf{for}\ (x \leftarrow M)\ N \mid \textbf{table}\ n\ \langle R\rangle$
		$\mid$	$\texttt{Lit}\ M \mid \texttt{If}\ M \mid \texttt{For}\ C\ M \mid \texttt{Cell}\ M \mid \texttt{OpEq}\ C\ M \mid \texttt{OpPlus}\ M$
		$\mid$	$\textbf{tracecase}\ M\ \textbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P)$
		$\mid$	$\textbf{typecase}\ C\ \textbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \beta.M_T)$

Figure 6.2: The syntax of LINKS^{T}.

```
tracecase M of
  Lit x      => M_L
  If x       => M_I
  For α x    => M_F
  Cell x     => M_C
  OpEq α x   => M_E
  OpPlus x   => M_P
```

Note that **typecase** takes extra arguments to deal with records and traces compared to the feature preview in Section 6.1.2.

The type T(*C*) in essence allows us to go from constructors to types. Constructors duplicate some types, but not all, and add some type-level computation constructs. We have base constructors, lists and records, which are all marked by a star that we will frequently omit if it does not matter whether we use a type or constructor or it is clear from the context. LINKS^{T}, unlike λ_i^{ML}, does not

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

Figure 6.3: Well-formed contexts Γ.

have a constructor for function types because we do not need it in queries—our main focus. A type variable is a constructor, introduced either by a polymorphic type, see above, or by a type-level function. Constructors also contain type-level function application and **`Typerec`**. Compared to the feature preview in Section 6.1.1, **`Typerec`** takes extra arguments for record and trace types.

Terms are mostly standard. Constants c are Boolean, number, and string literals. Functions and term-level type abstraction are unary. We only list the operators + and ==, other standard operators work similarly. Records are written in angle brackets, as are record types. Record map and record fold take a row constructor S as an argument. We write this in superscript to indicate that the generic record operations are specialized by that row, see Section 6.5 for details. The first term argument to **`rmap`** and **`rfold`** is a function. The last term argument is a record whose row matches the superscript. The middle argument to **`rfold`** is the initial accumulator. The constructors `Lit`, `If`, `For`, etc. belong to the `Trace` type. The trace constructors `For` and `OpEq` carry not only a subtrace, but also a type constructor. The Links$^{\mathsf{T}}$ core language does not have variants, at this time.

Figure 6.3 defines well-formed, unified contexts Γ as partial maps from term variables to types and from type variables to kinds. Figure 6.4 shows the kinding rules for constructors and rows of constructors. Similarly, Figure 6.5 shows the kinding rules for types and row types. Morally, *BaseType* is a subkind of *Type* and *BaseRow*, a row of base types, is a subkind of *Row*. We leave the subkinding relation implicit and mostly just use *BaseType* as a hint, like in the kinding rule for `Trace`*.

Constructors are equivalent according the rules in Figure 6.7 with type-level computation defined by the rules in Figure 6.6. Type-level computations include application of type-level functions, mapping of type-level functions over rows using **`Rmap`**, and folding over types using **`Typerec`**. The rules do not enforce any particular evaluation order and may reduce nondeterministically anywhere

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash \texttt{Bool}^* : \mathit{BaseType}} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \texttt{Int}^* : \mathit{BaseType}} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \texttt{String}^* : \mathit{BaseType}} \qquad \frac{\Gamma(\alpha) = K}{\Gamma \vdash \alpha : K}$$

$$\frac{\Gamma, \alpha : K_1 \vdash C : K_2}{\Gamma \vdash \lambda\alpha : K_1.C : K_1 \to K_2} \qquad \frac{\Gamma \vdash C : K_1 \to K_2 \qquad \Gamma \vdash D : K_1}{\Gamma \vdash C\ D : K_2} \qquad \frac{\Gamma \vdash C : \mathit{Type}}{\Gamma \vdash \texttt{List}^*\ C : \mathit{Type}}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash C : \mathit{Type} \qquad \Gamma \vdash C_B : K \qquad \Gamma \vdash C_I : K \qquad \Gamma \vdash C_S : K \\ \Gamma \vdash C_L : \mathit{Type} \to K \to K \qquad \Gamma \vdash C_R : \mathit{Row} \to \mathit{Row} \to K \qquad \Gamma \vdash C_T : \mathit{BaseType} \to K \to K\end{array}}{\Gamma \vdash \textbf{Typerec}\ C\ (C_B, C_I, C_S, C_L, C_R, C_T) : K}$$

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

Figure 6.4: Constructor and row constructor kinding.

$$\frac{\Gamma \vdash C : \mathit{Type}}{\Gamma \vdash \texttt{T}(C) : \mathit{Type}} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \texttt{Bool} : \mathit{BaseType}} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \texttt{Int} : \mathit{BaseType}} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \texttt{String} : \mathit{BaseType}}$$

$$\frac{\Gamma \vdash A : \mathit{BaseType}}{\Gamma \vdash A : \mathit{Type}} \qquad \frac{\Gamma, \alpha : K \vdash A : \mathit{Type} \qquad \alpha \notin \mathit{Dom}(\Gamma)}{\Gamma \vdash \forall\alpha : K.A : \mathit{Type}} \qquad \frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash B : \mathit{Type}}{\Gamma \vdash A \to B : \mathit{Type}}$$

$$\frac{\Gamma \vdash A : \mathit{Type}}{\Gamma \vdash \texttt{List}\ A : \mathit{Type}} \qquad \frac{\Gamma \vdash R : \mathit{Row}}{\Gamma \vdash \texttt{Record}\ R : \mathit{Type}} \qquad \frac{\Gamma \vdash A : \mathit{BaseType}}{\Gamma \vdash \texttt{Trace}\ A : \mathit{Type}} \qquad \frac{\Gamma \vdash S : \mathit{Row}}{\Gamma \vdash \texttt{T}(S) : \mathit{Row}}$$

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash \cdot : \mathit{Row}} \qquad \frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash R : \mathit{Row}}{\Gamma \vdash l^P : A; R : \mathit{Row}}$$

Figure 6.5: Type and row type kinding.

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

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

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

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

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

Figure 6.6: Constructor and row constructor computation.

$$\frac{\Gamma \vdash C : K}{\Gamma \vdash C = C : K} \qquad \frac{\Gamma \vdash D = C : K}{\Gamma \vdash C = D : K} \qquad \frac{\Gamma \vdash C = C' : K \qquad \Gamma \vdash C' = C'' : K}{\Gamma \vdash C = C'' : K}$$

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

$$\frac{\Gamma \vdash C = C' : K' \to K \qquad \Gamma \vdash D = D' : K'}{\Gamma \vdash C\ D = C'\ D' : K} \qquad \frac{\Gamma \vdash C = D : K}{\Gamma \vdash \texttt{List}^*\ C = \texttt{List}^*\ D : K}$$

$$\frac{\Gamma \vdash S = S' : K}{\Gamma \vdash \texttt{Record}^*\ S = \texttt{Record}^*\ S' : K} \qquad \frac{\Gamma \vdash C = D : K}{\Gamma \vdash \texttt{Trace}^*\ C = \texttt{Trace}^*\ D : K}$$

$$\frac{\Gamma \vdash C : K \qquad \Gamma \vdash D : K \qquad C \rightsquigarrow D}{\Gamma \vdash C = D : K} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \cdot = \cdot : Row}$$

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

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

Figure 6.7: Constructor and row constructor equivalence.

$$\frac{\Gamma \vdash A : K}{\Gamma \vdash A = A : K} \qquad \frac{\Gamma \vdash B = A : K}{\Gamma \vdash A = B : K} \qquad \frac{\Gamma \vdash A = A' : K \qquad \Gamma \vdash A' = A'' : K}{\Gamma \vdash A = A'' : K}$$

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{T}(\mathtt{Bool}^*) = \mathtt{Bool} : \mathit{BaseType}} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathtt{T}(\mathtt{Int}^*) = \mathtt{Int} : \mathit{BaseType}}$$

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

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

$$\frac{}{\Gamma \vdash T(\cdot) = \cdot : \mathit{Row}} \qquad \frac{\Gamma \vdash C : \mathit{Type} \qquad \Gamma \vdash S : \mathit{Row}}{\Gamma \vdash T(l^P : C; S) = (l^P : T(C); T(S)) : \mathit{Row}} \qquad \frac{\Gamma \vdash C = D : \mathit{Type}}{\Gamma \vdash \mathtt{T}(C) = \mathtt{T}(D) : \mathit{Type}}$$

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

$$\frac{\Gamma \vdash A = B : \mathit{BaseType}}{\Gamma \vdash \mathtt{Trace}\ A = \mathtt{Trace}\ B : \mathit{Type}} \qquad \frac{\Gamma \vdash A = A' : \mathit{Type} \qquad \Gamma \vdash B = B' : \mathit{Type}}{\Gamma \vdash A \to B = A' \to B' : \mathit{Type}}$$

$$\frac{\Gamma \vdash A = B : \mathit{Type}}{\Gamma \vdash \forall \alpha.A = \forall \alpha.B : \mathit{Type}} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \cdot = \cdot : \mathit{Row}} \qquad \frac{\Gamma \vdash A = B : \mathit{Type} \qquad \Gamma \vdash R = R' : \mathit{Row}}{\Gamma \vdash (l^P : A; R) = (l^P : B; R') : \mathit{Row}}$$

Figure 6.8: Type and row type equivalence.

inside a term. Note also that type equality is symmetric so we can go, for example, from α to $(\lambda\alpha.\alpha)\ \alpha$ by symmetry and the type application β-rule.

Type equivalence is defined by the rules in Figure 6.8. There is no type-level computation in types, it all happens in constructors, but the equivalence rules for type $\mathtt{T}(C)$ show that the results of type-level computations have equivalents in types. Note how unfinished type-level computations, in particular all the left-hand sides in Figure 6.6, and type-level computations stuck on type variables are not equivalent to any type.

The typing rules for terms are listed in Figure 6.9. We assume a signature Σ that contains the types of constants c. Unlike in Links, we do not treat primitive operators like + as constants. We list explicit typing rules for + and == as representative examples (one monomorphic, one polymorphic) of the other

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

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B} \qquad \frac{\Gamma, \alpha : K \vdash M : A \qquad \alpha \notin \mathit{Dom}(\Gamma)}{\Gamma \vdash \Lambda\alpha : K.M : \forall \alpha : K.A}$$

$$\frac{\Gamma \vdash M : \forall \alpha : K.A \qquad \Gamma \vdash C : K}{\Gamma \vdash M\ C : A[\alpha := C]} \qquad \frac{\Gamma \vdash A \qquad \Gamma, f : A \vdash M : A}{\Gamma \vdash \mathbf{fix}\ f : A.M : A}$$

$$\frac{\Gamma \vdash L : \mathtt{Bool} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash \mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N : A} \qquad \frac{\Gamma \vdash M : \mathtt{Int} \qquad \Gamma \vdash N : \mathtt{Int}}{\Gamma \vdash M + N : \mathtt{Int}}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A \qquad \Gamma \vdash A : \mathit{BaseType}}{\Gamma \vdash M == N : \mathtt{Bool}} \qquad \frac{\cdot \vdash (\mathsf{oid} : \mathtt{Int}, R) : \mathit{BaseRow}}{\Gamma \vdash \mathbf{table}\ n\ \langle \mathsf{oid} : \mathtt{Int}, R \rangle : \mathtt{List} \langle \mathsf{oid} : \mathtt{Int}, R \rangle}$$

$$\frac{\Gamma \vdash A : \mathit{Type}}{\Gamma \vdash [] : \mathtt{List}\ A} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : \mathtt{List}\ A} \qquad \frac{\Gamma \vdash M : \mathtt{List}\ A \qquad \Gamma \vdash N : \mathtt{List}\ A}{\Gamma \vdash M +\!\!+ N : \mathtt{List}\ A}$$

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

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

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

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

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

Figure 6.9: Term formation $\Gamma \vdash M : A$.

$$\frac{\Gamma \vdash c : \texttt{Bool}}{\Gamma \vdash \texttt{Lit}\ c : \texttt{Trace Bool}} \qquad \frac{\Gamma \vdash c : \texttt{Int}}{\Gamma \vdash \texttt{Lit}\ c : \texttt{Trace Int}} \qquad \frac{\Gamma \vdash c : \texttt{String}}{\Gamma \vdash \texttt{Lit}\ c : \texttt{Trace String}}$$

$$\frac{\Gamma \vdash M : \langle \mathsf{cond} : \texttt{Trace Bool}, \mathsf{out} : \texttt{Trace}\ A \rangle}{\Gamma \vdash \texttt{If}\ M : \texttt{Trace}\ A}$$

$$\frac{\Gamma \vdash C : \mathit{Type} \qquad \Gamma \vdash M : \langle \mathsf{in} : \texttt{T (TRACE}\ C\texttt{)}, \mathsf{out} : \texttt{Trace}\ A \rangle}{\Gamma \vdash \texttt{For}\ C\ M : \texttt{Trace}\ A}$$

$$\frac{\Gamma \vdash M : \langle \mathsf{table} : \texttt{String}, \mathsf{column} : \texttt{String}, \mathsf{row} : \texttt{Int}, \mathsf{data} : A \rangle}{\Gamma \vdash \texttt{Cell}\ M : \texttt{Trace}\ A}$$

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

$$\frac{\Gamma \vdash M : \langle \mathsf{left} : \texttt{Trace Int}, \mathsf{right} : \texttt{Trace Int} \rangle}{\Gamma \vdash \texttt{OpPlus}\ M : \texttt{Trace Int}}$$

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

Figure 6.10: `Trace` introduction and elimination rules.

arithmetic and relational operators. The rule for tables enforces that every table has an `oid` column. There is no separate type for tables, instead tables just have an appropriate list type. Unlike in LINKS, we do not distinguish comprehensions over tables and lists syntactically. The typing rules for **`rmap`** and **`rfold`** enforce that the annotated row constructor S matches the record type. The record fold rule additionally enforces that the record is homogeneous by requiring an **`Rmap`** type with a constant function.

The typing rules for `Trace` introduction and elimination in Figure 6.10 make use of the type-level function `TRACE` as defined in Figure 6.15. The `TRACE` function takes any query type and replaces all base types by their traced version. We list rules for the operators + and == only. `OpPlus` serves as an example for the other monomorphic operators. `OpEq` similarly serves as an example for the other polymorphic operators.

Type-checking of λ_i^{ML} is decidable [Crary et al., 2002; Morrisett, 1995]. We believe type-checking of LINKS^T is decidable, too. Type inference is an issue. UR/WEB has generic record programming features that make type inference undecidable in general, but Chlipala [2010] claims their heuristics make it still usable in practice. Maybe something similar would work for LINKS^T.

6.3 Recovering provenance from traces

In this section we demonstrate how to use LINKS^T to write trace analysis functions that extract provenance from traces. We start with the `value` function, which extracts the plain, unannotated value from a trace, then move on to where-provenance (Section 6.3.2) and lineage (Section 6.3.3).

6.3.1 Value

> Eventually I came to regard nondeterminacy as the normal situation, determinacy being reduced to a —not even very interesting— special case.
>
> E. W. Dijkstra
> A Discipline of Programming, 1976, page xv

The purpose of tracing queries is to extract provenance. However, we can use the same mechanism to extract the original values from a trace. Basically, applying the `value` trace analysis function to the trace of a query Q should have the same result as Q itself without any tracing.

On the level of types we have that tracing a query Q of type C results in a trace of type $\texttt{TRACE}\ C$. The type-level function `VALUE` in Figure 6.12 undoes what `TRACE` does. On base types it is the identity; on list and record types it recursively applies itself to element and field types; and on trace types it removes the type constructor `Trace`. In other words `VALUE` is the left inverse of `TRACE` on nested relational query types.

Lemma 6.11. *For all query type constructors C and row constructors S and well-formed contexts Γ:*

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

and

$$\Gamma \vdash \textbf{Rmap}\ \texttt{VALUE}\ (\textbf{Rmap}\ \texttt{TRACE}\ S) = S$$

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

- Base types $\texttt{Bool}^*$, $\texttt{Int}^*$, $\texttt{String}^*$:

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

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

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

- 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 $\cdot$: $\textbf{Rmap}\ \texttt{VALUE}\ (\textbf{Rmap}\ \texttt{TRACE}\ \cdot) = \cdot$

```
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
```

Figure 6.12: The `value` trace analysis function and `VALUE` type-level function.

- Row cons $(l : A, S)$:

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

□

The `value` function is shown in Figure 6.12. It has the polymorphic type `∀a.a -> VALUE a`. Thus, when we apply it to the type `TRACE` C we get a function from `TRACE` C to `VALUE (TRACE` C`)` which is the same as C, exactly as expected.

The implementation of `value` is by **`typecase`**. For the base types nothing needs to be done and `value` behaves like the identity. In fact, when `value` is

applied to a trace generated by Links^{T}, these cases will never be encountered. To obtain a list of values from a list of *b*s we traverse the list using a **`for`** comprehension and recursively call the `value` function at the element type `b`. Similarly for records we map the `value` function over the record fields. This is where the first of the generic record programming features of Links^{T} comes in. The new keyword **`rmap`** calls `value` with the type and value of every field in the record `x` and collects the result in a new record. Upon reaching an actual trace we case-split on the constructor using **`tracecase`**. Literals carry their value directly. `If` and `For` nodes both carry a subtrace of the same type so we recursively call `value` on that. A database `Cell` carries its value in the `data` field. Finally, the various operator traces carry subtraces on which we recursively call `value` and then combine the results using the appropriate operator.

The `value` function is structurally recursive.

6.3.2 Where-provenance

This section describes how to extract where-provenance from a trace. With Links^{W} we explored type system extensions to make handling where-provenance safer and give static guarantees about the presence of useful annotations. Here we use a more traditional approach in which where-provenance annotations are just more data, like lineage in Links^{L}.

We replace every cell in the output with a record of the original data as well as table, column, and row number. On the type-level this is done by `WHERE` defined in Figure 6.13. Base types in the result are replaced by record types using the helper function `W`. Like in `VALUE` before, lists and records are just traversed recursively. Traces are replaced by `W`. For example:

```
  WHERE (TRACE [⟨a: Int, b: [String]⟩])
= WHERE [⟨a: Trace Int, b: [Trace String]⟩]
= [⟨a: W Int, b: [W String]⟩]
= [⟨a: ⟨data: Int, table:String, column:String, row:Int⟩,
    b: [⟨data: String, table:String, column:String, row:Int⟩]⟩]
```

The trace analysis function `wherep` itself is defined in Figure 6.13 and has type `∀a.T(TRACE a) -> T(WHERE a)`. The overall structure is much the same as the `value` function before: we define a recursive, polymorphic function by cases on the argument type. The base cases are unreachable because a trace will always have leaves of `Trace` types, see Lemma 6.16. Therefore it does not

```
W = λa:Type.⟨data:a, table:String, column: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
    Bool     => fake Bool
    Int      => fake Int
    String   => fake String
    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) y.left +
                            value (Trace Int) y.right)
      OpEq c y => fake Bool (value (TRACE c) y.left ==
                             value (TRACE c) y.right)

fake : ∀a.T(a) -> T(W a)
fake = Λa:Type.λx:T(a).⟨data = x, table = "facts",
                        column = "alternative", row = -1⟩
```

Figure 6.13: The `wherep` trace analysis function and supporting definitions.

matter what the implementation is, as long as it typechecks. The helper function `fake` annotates a value with a fake annotation—something like $\bot$. For lists and records we map `wherep` over the elements and fields, respectively. Traces of literals are annotated with a fake annotation to indicate that they do not come from the database. As before, the cases for `If` and `For` annotate their output. The case for a database cell is the most important, yet trivial: The `Cell` trace constructor already carries value and where-provenance in exactly the right format, so we just return it. Finally, the operators use the `value` function to compute values from subtraces, perform the appropriate operation and annotate the result with a fake annotation.

This implementation demonstrates that it is possible to compute where-provenance using this tracing approach. It is easy to see how we could use Links$^{\mathrm{T}}$ to implement further variants of where-provenance at least as flexible as the limited user-defined annotations in Links$^{\mathrm{W}}$. It is not at all clear how to recover the strong typing discipline for provenance annotations that Links$^{\mathrm{W}}$ provides. We currently do not have a proof that the annotations are the same as those that Links$^{\mathrm{W}}$ would have produced. We tested a number of small examples and so far the generated queries look very similar, see Section 6.6 for one example.

6.3.3 Lineage

As far as possible, this implementation of lineage aims to emulate the behavior of Links$^{\mathrm{L}}$ as described in Chapter 4. This is slightly challenging, because 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 leaves and pull them up to the nearest enclosing list constructor.

On the type-level, we have the `LINEAGE` function as defined in Figure 6.14. The purpose of `LINEAGE` is the same as that of the lineage type translation $\mathfrak{L}$ from Chapter 4 defined in Figure 4.2 on page 59. Given a query type it places an annotation on every list element type. For example:

```
  LINEAGE [⟨a: [Int]⟩]
= [L ⟨a: [L Int]⟩]
= [⟨data: ⟨a: [⟨data: Int,
                lineage: [⟨table: String, row: Int⟩]⟩]⟩,
    lineage: [⟨table: String, row: Int⟩]⟩]
```

We have $\forall\alpha : \mathit{Type}.\texttt{LINEAGE}\ (\texttt{TRACE}\ \alpha) = \texttt{LINEAGE}\ \alpha$. The only interesting case is the one for record types, where we apply the induction hypothesis on the composition of type functions in a row map operation on a smaller type ρ: **Rmap** $(\texttt{LINEAGE} \cdot \texttt{TRACE})\ \rho = $ **Rmap** $\texttt{LINEAGE}\ \rho$.

On the value-level the implementation is split into two functions: `lineage` and `linnotation`, as shown in Figure 6.14. The `lineage` function matches on the type of its argument and makes (recursive) calls to `lineage`, `linnotation`, and `value` as appropriate. The `linnotation` function does the actual work of computing lineage annotations from traces. It assigns an empty list of annotations to base types (again, these are unreachable for traced queries). The case for lists concatenates the lineage annotations obtained by calling `linnotation` on the list elements. The case for records does the same, but in a slightly more roundabout way: 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.[3] 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, c: String⟩) [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, c: String⟩)
  [⟨data = x.a,
    lineage = [⟨table = "xs", row = x.oid⟩] ++
              [⟨table = "xs", row = x.oid⟩] ++
              [⟨table = "xs", row = x.oid⟩] ++
              [⟨table = "xs", row = x.oid⟩]⟩]
```

[3] The attentive reader might remember that we previously pointed out that it is best to use a commutative combining function in **rfold** because record labels are unordered. Here we use list concatenation which is decidedly non-commutative. More on this in a bit.

```
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
    Bool   => id a  -- unreachable
    Int    => id a  -- unreachable
    String => id a  -- unreachable
    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
    Bool | Int | String => λ_.[] -- unreachable
    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 6.14: The `lineage` trace analysis function and supporting definitions.

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 four times. In fact, a similar query on a table with n columns, would produce $n+1$ annotations. Looking at the (normalized) traced expression below, we can see the problem.

```
for (x <- table "xs" ⟨a: Int, b: Bool, c: String⟩)
  [For ⟨in=⟨a=Row ⟨table="xs", column="a", row=x.oid, data=x.a⟩,
            b=Row ⟨table="xs", column="b", row=x.oid, data=x.b⟩,
            c=Row ⟨table="xs", column="c", row=x.oid, data=x.c⟩⟩,
       out=Row ⟨table="xs", column="a", row=x.oid, data=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. Our definition of lineage from Chapter 4 is really about rows, not cells. This clashes with our traces, which are on cells, not rows. There are at least two solutions to this problem that preserve tracing at the level of cells. The ad-hoc solution is not optimal in general, but would be sufficient here, and maybe other cases: We could 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, at query normalization time. In the example above we would only need to test alpha equivalence, but other queries and other forms of provenance might require more sophisticated analysis for determining duplicates. 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.

6.3.4 Other forms of provenance

Following the same approach, it should be possible to implement other forms of provenance based on annotation propagation, such as the cell-level lineage of Müller et al. [2018]. Dependency provenance [Cheney et al., 2011] has annotations on multiple levels which may be difficult to recover from cell-level traces alone. Semiring provenance is another interesting candidate. One challenge is, again, that it is defined on rows. One opportunity is that trace analysis functions are just that, functions, so a generic semiring trace analysis function could be parameterized by concrete $+$ and $\times$ operations and 0 and 1 values.

Expression provenance [Acar et al., 2012] annotates values with a tree of abstract expressions that were executed to obtain the value. Conceptually this is very close to the trace itself but the `Trace` type is not a query type. To implement

```
TRACE = λa:Type. Typerec a (Trace Bool, Trace Int, Trace String,
                            λ _ b. List b, λ _ r. Record r, λ _ b. b)
```

Figure 6.15: The type-level function `TRACE`.

expression provenance in Links^T we would need to find a suitable (nested) relational encoding. Giorgidze et al. [2013] show how to encode arbitrary nonrecursive algebraic datatypes relationally and use this in an extension to the language-integrated query library Dsh. The datatype for expressions is recursive, but of a limited depth and thus can be unrolled. It seems possible to do all of this in a trace analysis function, but perhaps it would be more fruitful to extend Links^T itself to deal with algebraic datatypes to make this easier. Unfortunately, even full support for recursive algebraic datatypes is not quite enough to define the `Trace` type in the language, because that also needs to store type information for polymorphic operations and comprehensions.

Combining forms of provenance is another interesting exercise. We have already seen multiple trace analysis functions used together: both `wherep` and `lineage` use `value`. However, it would be ill-typed to call `wherep` and then `lineage` on a query trace, because `wherep` does not preserve the trace. Manually fusing the where-provenance and lineage analysis functions is certainly possible, but somewhat dissatisfying. It would be interesting to see if they can be written in a way that preserves a copy of the trace, so that multiple trace analysis functions could be stacked.

Fortunately, it is not necessary to extend Links^T to define new forms of provenance or provenance combinators, which should make it much easier to carry out such experiments in the future.

6.4 Self-tracing queries

This section describes the transformation that turns a normalized query Q_n into a query Q_t that, when executed, returns a trace of the execution of Q_n.

As mentioned before, the most important aspect of our notion of traces is that all the tracing information is at the level of base types. An annotated value of base type A has type `Trace` A. Records and lists on the other hand do not carry any annotations themselves. In other words, the trace of a list is a list

of traces and the trace of a record is a record of traces. A query typically has a larger type than just a base type, at least a list constructor. The type-level function `TRACE`, defined in Figure 6.15, describes the type of a traced query. If a query Q_n has type `T`(C), its traced version Q_t has type `T(TRACE` C`).` `TRACE` is defined in terms of **`Typerec`** to recursively traverse a tree made of `List` and `Record` constructors and apply the `Trace` type constructor to all the base types at the leaves of the tree. For example:

```
  TRACE [⟨a: [Int], b: ⟨c: String⟩⟩]
= [TRACE ⟨a: [Int], b: ⟨c: String⟩⟩]
= [⟨a: TRACE [Int], b: TRACE ⟨c: String⟩⟩]
= [⟨a: [TRACE Int], b: ⟨c: TRACE String⟩⟩]
= [⟨a: [Trace Int], b: ⟨c: Trace String⟩⟩]
```

Lemma 6.16. *For all query types C,* `TRACE` C *is not a base type.*

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. □

Figure 6.17 shows the self-tracing transformation $[\![\cdot]\!]$ and two helper functions. The self-tracing transformation takes a term of type `T(`C`)` and returns a self-tracing query of type `T(TRACE` C`)` for any query type C. The *dist* function is a meta-level helper function that distributes a trace constructor over lists and records. 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 all the leaves and wraps the expression with the hole around them.[4] We need it to push down trace information over list and record constructors and into the leaves.

We apply the tracing transformation to normalized queries. We do not rely on the normal form itself, but only on the absence of non-primitive functions, variants, and free variables. Thus, all variables that appear in a query are introduced by for-comprehensions.

[4] We could write *dist* as an object-level function with the type below, but going from what is basically a partially applied `Trace` constructor to a fully polymorphic function involves a lot of boilerplate code for handling impossible non-base-type cases.

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

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![c]\!] &= \texttt{Lit}\ c \\
[\![\textbf{if}\ L\ \textbf{then}\ M\ \textbf{else}\ N : \mathsf{T}(C)]\!] &= \textbf{if}\ \texttt{value (Trace Bool)}\ [\![L]\!] \\
&\quad\ \textbf{then}\ \mathit{dist}(\texttt{TRACE}\ C, \texttt{If}\ \langle \texttt{cond} = [\![L]\!], \texttt{out} = \mathbb{H}\rangle, [\![M]\!]) \\
&\quad\ \textbf{else}\ \mathit{dist}(\texttt{TRACE}\ C, \texttt{If}\ \langle \texttt{cond} = [\![L]\!], \texttt{out} = \mathbb{H}\rangle, [\![N]\!]) \\
[\![[]]\!] &= [] \\
[\![[M]]\!] &= [[\![M]\!]] \\
[\![M +\!\!+ N]\!] &= [\![M]\!] +\!\!+ [\![N]\!] \\
[\![\textbf{for}\ (x \leftarrow M : D)\ N : \mathsf{T}(C)]\!] &= \textbf{for}\ (x \leftarrow [\![M]\!]) \\
&\quad\ \mathit{dist}(\texttt{TRACE}\ C, \texttt{For}\ D\ \langle \texttt{in} = x, \texttt{out} = \mathbb{H}\rangle, [\![N]\!]) \\
[\![\langle \overline{l = M} \rangle]\!] &= \langle \overline{l = [\![M]\!]} \rangle \\
[\![M.l]\!] &= [\![M]\!].l \\
[\![\textbf{table}\ n\ \langle \overline{l : C} \rangle]\!] &= \textbf{for}\ (y \leftarrow \textbf{table}\ n\ \langle \overline{l : C} \rangle) \\
&\quad\ [\langle \overline{l = \mathit{cell}(n, l, y.\texttt{oid}, y.l)} \rangle] \\
[\![M == (N : \mathsf{T}(C))]\!] &= \texttt{OpEq}\ C\ \langle \texttt{left} = [\![M]\!], \texttt{right} = [\![N]\!] \rangle \\
[\![M + N]\!] &= \texttt{OpPlus}\ \langle \texttt{left} = [\![M]\!], \texttt{right} = [\![N]\!] \rangle \\
\\
\mathit{cell}(t, c, r, d) &= \texttt{Cell}\ \langle \texttt{table} = t, \texttt{column} = c, \texttt{row} = r, \texttt{data} = d \rangle \\
\\
\mathit{dist}(\texttt{Trace}\ C, k, t) &= k[\mathbb{H} := t] \\
\mathit{dist}(\texttt{List}\ C, k, l) &= \textbf{for}\ (x \leftarrow l)\ [\mathit{dist}(C, k, x)] \\
\mathit{dist}(\langle \overline{l : C} \rangle, k, r) &= \langle \overline{l = \mathit{dist}(C, k, r.l)} \rangle
\end{aligned}
$$

Figure 6.17: Self-tracing transformation.

Variables are left untouched by the transformation, but in the self-tracing query they have different types, namely trace types, as seen in the translation of for-comprehensions. Literal values in the query are annotated with the `Lit` constructor. To trace if-then-else we need to trace both branches and return the correct one depending on whether the condition holds at runtime. We use the trace analysis function `value` (defined in Section 6.3.1) to turn the trace of the condition, which has type `Trace Bool`, back into a plain unannotated `Bool`. This simplest of the trace analysis functions discards all trace annotations and is described in more detail in Section 6.3.1. We cannot just *not* trace the condition, because it may contain bound variables, but their type is different in

the self-tracing query. We use the *dist* function to distribute an annotation over both branches that also contains the trace of the conditional. The cases for empty lists, singleton lists, and list concatenation are the obvious ones considering that a trace of a list is a list of traces. Self-tracing for-comprehensions iterate over traced input and annotate every output element with the corresponding input, again using *dist*. Note that as mentioned before the variables bound by comprehensions have different types in the original and self-tracing query. Record construction and projection are again the obvious translations considering that a trace of a record is a record of traces. Tables are translated to iterations over the tables that provide initial annotations, similar to the where-provenance and lineage transformations earlier. Here the initial annotation uses the *cell* function which is just the `Cell` constructor in disguise. It takes the familiar annotations table name, column name, row number via `oid`, and the data itself. Operators like == and + record subtraces for their arguments. Equality, as a polymorphic operation, additionally records the type at which it was applied. We need this later when analyzing a trace.

As a sanity check we prove that the result of the tracing transformation is type correct. We prove a lemma about *dist* first.

Lemma 6.18 (Distributing trace constructors is type correct)**.** *For all type constructors C that are equivalent to a query result type (base types, list types, closed record types, see Figure 6.36) and 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*, $dist($`TRACE` $C, k, M)$ has type* `TRACE` C*.*

Proof. By induction on the query type constructor C.

- The base cases are `Bool`, `Int`, and `String`. For any base type O out of these, we have `TRACE` $O =$ `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 6.25).

- Case $C =$ `List` (`TRACE` C'): We need the right-hand side

  $$\textbf{for}\ (x \leftarrow l)\ [dist(\texttt{TRACE}\ C', k, x)]$$

to have type TRACE (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 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 TRACE C' which they do by induction hypothesis. □

We only trace closed terms. However, we cannot require an empty context, because then the induction would not work out for comprehensions. Therefore we define a tracing transformation on contexts as follows.

Definition 6.19 (Trace context)**.** $[\![\Gamma]\!]$ maps term variable x to T(TRACE C) if and only if Γ maps x to A, where C is the obvious constructor with $\cdot \vdash A = \texttt{T}(C)$.

Lemma 6.20. *For every query type A made of base types, list constructors, and closed records, there exists C such that $\Gamma \vdash A = \texttt{T}(C)$ in a well-formed context Γ.*

Theorem 6.21 (Trace rewriting is type correct)**.** *If $\Gamma \vdash M : A$ then for all C, if $\Gamma \vdash A = \texttt{T}(C)$ then $[\![\Gamma]\!] \vdash [\![M]\!] : \texttt{T}(\texttt{TRACE}\ C)$, 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.*

Proof. By induction on the typing derivation for $M : \texttt{T}(C)$. Almost all cases require that some subterms have a type $\texttt{T}(C')$ that is equal to some query type A. We can obtain this constructor C' by Lemma 6.20.

- Case $\dfrac{\Gamma(x) = A}{\Gamma \vdash x : A}$: $\dfrac{[\![\Gamma]\!](x) = \texttt{T}(\texttt{TRACE}\ C) \quad \text{(Definition 6.19)}}{[\![\Gamma]\!] \vdash x : \texttt{T}(\texttt{TRACE}\ C)}$

- Literals c have base types Bool, Int, or String. Their traces Lit c have types Trace Bool, Trace Int, or Trace String, respectively.

- Case $\dfrac{\Gamma \vdash L : \texttt{Bool} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash \textbf{if}\ L\ \textbf{then}\ M\ \textbf{else}\ N : A}$:

The right hand side of the self-tracing transform is another **`if-then-else`** with condition `value (Trace Bool)` $[\![L]\!]$ and then-branch

$$\mathit{dist}(\texttt{TRACE}\ C, \texttt{If}\ \langle \texttt{cond} = [\![L]\!], \texttt{out} = \mathbb{H}\rangle, [\![M]\!])$$

and similar else-branch.

In the condition, we apply $\texttt{value} : \forall\alpha.\texttt{T}(\alpha) \to \texttt{T}(\texttt{VALUE}\ \alpha)$ to a subtrace of type `TRACE Bool` by induction hypothesis. Therefore it has type `VALUE (TRACE Bool)` which is equal to `Bool` by Lemma 6.11.

For all base types D, $\texttt{If}\ \langle \texttt{cond} = [\![L]\!], \texttt{out} = \mathbb{H}\rangle$ has type $\texttt{Trace}\ D$ assuming $\mathbb{H} : \texttt{Trace}\ D$. We have $[\![M]\!] : \texttt{T}(\texttt{TRACE}\ C)$ by IH. Therefore, by Lemma 6.18, the whole term obtained by *dist* has type `TRACE` C. The else-branch is analogous and the whole expression has type $\texttt{T}(\texttt{TRACE}\ C)$.

- Case $\dfrac{}{\Gamma \vdash [] : \texttt{List}\ A}$:

$$\dfrac{\dfrac{[\![\Gamma]\!] \vdash \texttt{T}(\texttt{TRACE}\ C) : \mathit{Type}\ \text{using}\ A = \texttt{T}(C)}{[\![\Gamma]\!] \vdash [] : \texttt{List}\ \texttt{T}(\texttt{TRACE}\ C)}}{[\![\Gamma]\!] \vdash [] : \texttt{T}(\texttt{List}^*\ (\texttt{TRACE}\ C))}}{[\![\Gamma]\!] \vdash [] : \texttt{T}(\texttt{TRACE}\ (\texttt{List}^*\ C))}$$

- Case $\dfrac{\Gamma \vdash M : A}{\Gamma \vdash [M] : \texttt{List}\ A}$:

$$\dfrac{\dfrac{\dfrac{\text{IH}}{[\![\Gamma]\!] \vdash [\![M]\!] : \texttt{T}(\texttt{TRACE}\ C)}}{[\![\Gamma]\!] \vdash [[\![M]\!]] : \texttt{List}\ \texttt{T}(\texttt{TRACE}\ C)}}{[\![\Gamma]\!] \vdash [[\![M]\!]] : \texttt{T}(\texttt{TRACE}\ (\texttt{List}^*\ C))}$$

- Case $\dfrac{\Gamma \vdash M : \texttt{List}\ A \qquad \Gamma \vdash N : \texttt{List}\ A}{\Gamma \vdash M +\!\!+ N : \texttt{List}\ A}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{IH}}{[\![\Gamma]\!] \vdash [\![M]\!] : \texttt{T}(\texttt{TRACE}\ (\texttt{List}^*\ C))}}{[\![\Gamma]\!] \vdash [\![M]\!] : \texttt{List}\ \texttt{T}(\texttt{TRACE}\ C)} \qquad \text{analogous for } N}{[\![\Gamma]\!] \vdash [\![M]\!] +\!\!+ [\![N]\!] : \texttt{List}\ \texttt{T}(\texttt{TRACE}\ C)}}{[\![\Gamma]\!] \vdash [\![M]\!] +\!\!+ [\![N]\!] : \texttt{T}(\texttt{TRACE}\ (\texttt{List}^*\ C))}$$

- Case $\dfrac{\Gamma \vdash M : \texttt{List}\ B \qquad \Gamma, x : B \vdash N : \texttt{List}\ A}{\Gamma \vdash \textbf{for}\ (x \leftarrow M)\ N : \texttt{List}\ A}$:

$$\dfrac{\dfrac{\dfrac{\text{IH}}{[\![\Gamma]\!] \vdash [\![M]\!] : \texttt{T}(\texttt{TRACE}\ (\texttt{List}^*\ D))}}{[\![\Gamma]\!] \vdash [\![M]\!] : \texttt{List}\ \texttt{T}(\texttt{TRACE}\ D)} \qquad \dfrac{\star}{[\![\Gamma]\!], x : \texttt{T}(\texttt{TRACE}\ D) \vdash b : \texttt{List}\ \texttt{T}(\texttt{TRACE}\ C)}}{[\![\Gamma]\!] \vdash \textbf{for}\ (x \leftarrow [\![M]\!])\ b : \texttt{List}\ \texttt{T}(\texttt{TRACE}\ C)}}{[\![\Gamma]\!] \vdash \textbf{for}\ (x \leftarrow [\![M]\!])\ b : \texttt{T}(\texttt{TRACE}\ (\texttt{List}^*\ C))}$$

  where $b = \mathit{dist}(\texttt{TRACE}\ C, \texttt{For}\ D\ \langle \texttt{in} = x, \texttt{out} = \mathbb{H} \rangle, [\![N]\!])$ and $\star$ follows from the induction hypothesis applied to $[\![N]\!]$ and Lemma 6.18.

- 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 $A = \texttt{T}(\texttt{TRACE}\ C)$ the trace of M contains label l with type $\texttt{TRACE}\ C$.

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

- 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.

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

  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:

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

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

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

□

6.5 Normalization

Our ultimate goal is to translate LINKS^{T} queries — including provenance extraction by trace analysis — to SQL. In Section 6.5.1 we define the reduction relation $\rightsquigarrow$ on LINKS^{T} terms, constructors, and row constructors. In Section 6.5.2, we show that this reduction relation preserves types. Section 6.5.3 describes the LINKS^{T} normal form. In Section 6.5.4 we prove a progress lemma: well-typed terms reduce or are in LINKS^{T} normal form. This is slightly different from standard progress in that the normal form is not a *value*. Rather, the normal form describes partially evaluated programs that cannot reduce further, because they are stuck on either database table references, or free variables. Progress and preservation imply the existence of a partial normalization function. Section 6.5.5 tightens the normal form for queries. Closed terms with query type, if they normalize at all, normalize to a term in the nested relational calculus. From there we can use previous work, for example query shredding [Cheney et al., 2014c] or flattening [Ulrich and Grust, 2015], to go the rest of the way to flat SQL queries.

6.5.1 Reduction relation

The reduction relation $\rightsquigarrow$ is overloaded for constructors, row constructors, and terms. We have already seen the reduction rules for constructors and row constructors in Figure 6.6. The reduction rules for terms are inspired by previous work on Links query normalization [Cheney et al., 2014c; Cooper, 2009; Lindley and Cheney, 2012].

Reduction rules fall into one of three categories: β-rules are listed in Figure 6.22 and perform computation when introduction and elimination forms meet, like application of a function, or projection out of a record; commuting conversions, listed in Figure 6.23, reorder terms to expose more β-reductions; and congruence rules (Figure 6.24) allow subterms to reduce independently. For example, `(if x then ⟨a = 5⟩ else y).a` reduces by a commuting conversion to `if x then ⟨a = 5⟩.a else y.a`. A β-rule reduces `⟨a = 5⟩.a` to `5` and a congruence rule allows this reduction to happen inside the **then** branch. Thus the whole term reduces to `if x then 5 else y.a`.

Unlike in standard evaluation rules, we perform reduction under binders, like functions and comprehensions and, as seen above, in the branches of if-then-else. This is crucial for removing all occurrences of language constructs that we cannot translate to Sql because comprehension bodies and conditionals can of course remain in the result of normalization.

The reduction rules do not impose any particular order of execution, like call-by-name or call-by-value, but since all operations are pure, this is not observable.

6.5.2 Preservation

To prove preservation we will need the following substitution lemmas. Substitution of variables in terms (1), type variables in types (2), and type variables in terms (4) are standard for λ_i^{ML} [Crary et al., 2002; Morrisett, 1995]. We add variants for row constructors: substitution of row variables in types (3), a variant of (2); and substitution of row variables in terms (5), a variant of (4). The row variants use the syntactic sort S in place of C and ρ instead of α.

Lemma 6.25 (Substitution lemmas)**.**

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

$$(\lambda x.M)\ N \rightsquigarrow M[x := N]$$

$$\mathbf{fix}\ f.M \rightsquigarrow M[f := \mathbf{fix}\ f.M]$$

$$(\Lambda\alpha.M)\ C \rightsquigarrow M[\alpha := C]$$

$$\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ M\ \mathbf{else}\ N \rightsquigarrow M$$

$$\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ M\ \mathbf{else}\ N \rightsquigarrow N$$

$$\langle \overline{l_i = M_i} \rangle.l_i \rightsquigarrow M_i$$

$$\mathbf{rmap}^{(\overline{l_i:C_i})}\ M\ N \rightsquigarrow \langle \overline{l_i = (M\ C_i)\ N.l_i} \rangle$$

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

$$\mathbf{for}\ (x \leftarrow [M])\ N \rightsquigarrow N[x := M]$$

$$\mathbf{tracecase}\ \mathtt{Lit}\ M\ \mathbf{of}\ (x.M_L, M_I, M_F, M_C, M_E, M_P) \rightsquigarrow M_L[x := M]$$

$$\mathbf{tracecase}\ \mathtt{If}\ M\ \mathbf{of}\ (M_L, x.M_I, M_F, M_C, M_E, M_P) \rightsquigarrow M_I[x := M]$$

$$\mathbf{tracecase}\ \mathtt{For}\ C\ M\ \mathbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P) \rightsquigarrow M_F[\alpha := C, x := M]$$

$$\mathbf{tracecase}\ \mathtt{Cell}\ M\ \mathbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P) \rightsquigarrow M_C[x := M]$$

$$\mathbf{tracecase}\ \mathtt{OpEq}\ C\ M\ \mathbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P) \rightsquigarrow M_E[\alpha := C, x := M]$$

$$\mathbf{tracecase}\ \mathtt{OpPlus}\ M\ \mathbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P) \rightsquigarrow M_P[x := M]$$

$$\mathbf{typecase}\ \mathtt{Bool}\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) \rightsquigarrow M_B$$

$$\mathbf{typecase}\ \mathtt{Int}\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) \rightsquigarrow M_I$$

$$\mathbf{typecase}\ \mathtt{String}\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) \rightsquigarrow M_S$$

$$\mathbf{typecase}\ \mathtt{List}\ C\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) \rightsquigarrow M_L[\beta := C]$$

$$\mathbf{typecase}\ \mathtt{Record}\ S\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) \rightsquigarrow M_R[\rho := S]$$

$$\mathbf{typecase}\ \mathtt{Trace}\ C\ \mathbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) \rightsquigarrow M_T[\gamma := C]$$

Figure 6.22: Normalization β-rules. See also commuting conversions in Figure 6.23, congruence rules in Figure 6.24, and constructor computation rules in Figure 6.6.

$$(\texttt{if}\ L\ \texttt{then}\ M_1\ \texttt{else}\ M_2)\ N \rightsquigarrow \texttt{if}\ L\ \texttt{then}\ M_1\ N\ \texttt{else}\ M_2\ N$$

$$(\texttt{if}\ L\ \texttt{then}\ M_1\ \texttt{else}\ M_2)\ C \rightsquigarrow \texttt{if}\ L\ \texttt{then}\ M_1\ C\ \texttt{else}\ M_2\ C$$

$$(\texttt{if}\ L\ \texttt{then}\ M\ \texttt{else}\ N).l \rightsquigarrow \texttt{if}\ L\ \texttt{then}\ M.l\ \texttt{else}\ N.l$$

$$\texttt{for}\ (x \leftarrow [])\ N \rightsquigarrow []$$

$$\texttt{for}\ (x \leftarrow M_1 +\!\!+ M_2)\ N \rightsquigarrow (\texttt{for}\ (x \leftarrow M_1)\ N) +\!\!+ (\texttt{for}\ (x \leftarrow M_2)\ N)$$

$$\texttt{for}\ (x \leftarrow \texttt{for}\ (y \leftarrow L)\ M)\ N \rightsquigarrow \texttt{for}\ (y \leftarrow L)\ \texttt{for}\ (x \leftarrow M)\ N$$

$$\texttt{if}\ (\texttt{if}\ L\ \texttt{then}\ M_1\ \texttt{else}\ M_2)\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$
$$\rightsquigarrow \texttt{if}\ L\ \texttt{then}\ (\texttt{if}\ M_1\ \texttt{then}\ N_1\ \texttt{else}\ N_2)\ \texttt{else}\ (\texttt{if}\ M_2\ \texttt{then}\ N_1\ \texttt{else}\ N_2)$$

$$\texttt{for}\ (x \leftarrow \texttt{if}\ L\ \texttt{then}\ M_1\ \texttt{else}\ M_2)\ N$$
$$\rightsquigarrow \texttt{if}\ L\ \texttt{then}\ \texttt{for}\ (x \leftarrow M_1)\ N\ \texttt{else}\ \texttt{for}\ (x \leftarrow M_2)\ N$$

$$\texttt{tracecase}\ \texttt{if}\ L\ \texttt{then}\ M_1\ \texttt{else}\ M_2\ \texttt{of}\ (M_L, M_I, M_F, M_C, M_E, M_P)$$
$$\rightsquigarrow \texttt{if}\ L\ \texttt{then}\ \texttt{tracecase}\ M_1\ \texttt{of}\ (M_L, M_I, M_F, M_C, M_E, M_P)$$
$$\texttt{else}\ \texttt{tracecase}\ M_2\ \texttt{of}\ (M_L, M_I, M_F, M_C, M_E, M_P)$$

Figure 6.23: Commuting conversions reorder expressions to expose more β-reductions.

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

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

Figure 6.24: Congruence rules allow subterms to reduce independently.

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* $\Gamma[\rho := S] \vdash M[\rho := S] : A[\rho := S]$.

We also need standard context manipulation lemmas for weakening and swapping the order of unrelated variables.

Lemma 6.26 (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 6.27 (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* $\Gamma, \beta : K_\beta, \alpha : K_\alpha \vdash B : K_B$.

With these helper lemmas out of the way, we can prove that the reduction relation $\rightsquigarrow$ preserves kinds of constructors and types of terms.

Lemma 6.28 (Constructor preservation)**.** *For all* Links$^{\mathrm{T}}$ *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* $\Gamma \vdash S' : K$.

Proof. By induction on the kinding derivation. We look at the possible reductions (see Figure 6.6). 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 6.25.
- **Rmap** $C \cdot \rightsquigarrow \cdot$: both sides have kind *Row*.
- **Rmap** $C\ (l^P : D; S) \rightsquigarrow (l^P : C\ D; \textbf{Rmap}\ C\ S)$: 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.
  - List left-hand side **Typerec** `List`* $D\ (C_B, C_I, C_S, C_L, C_R, C_T)$ and right-hand side $C_L\ D\ (\textbf{Typerec}\ D\ (C_B, C_I, C_S, C_L, C_R, C_T))$:

    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.
  - Record left-hand side **Typerec** `Record`* $S\ (C_B, C_I, C_S, C_L, C_R, C_T)$ and right-hand side $C_R\ S\ (\textbf{Rmap}\ (\lambda\alpha.\textbf{Typerec}\ \alpha\ (C_B, C_I, C_S, C_L, C_R, C_T))\ S)$:

    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. □

Lemma 6.29 (Preservation)**.** *For all* Links^{T} *terms* M *and* M'*, contexts* Γ*, and types* A*, if* $\Gamma \vdash M : A$ *and* $M \rightsquigarrow M'$*, then* $\Gamma \vdash M' : A$*.*

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 6.28, and congruence rules (see Figure 6.24), like $M + N$ being able to reduce in both M and N. The remaining, interesting reduction rules are the β-rules in Figure 6.22 and the commuting conversions in Figure 6.23. We discuss them grouped by the relevant typing rule.

- Function application:
  - $(\lambda x.M)\ N \rightsquigarrow M[x := N]$: follows from Lemma 6.25.
  - $(\textbf{if}\ L\ \textbf{then}\ M_1\ \textbf{else}\ M_2)\ N \rightsquigarrow \textbf{if}\ L\ \textbf{then}\ M_1\ N\ \textbf{else}\ M_2\ N$:

We have:

$$\dfrac{\dfrac{\Gamma \vdash L : \texttt{Bool} \qquad \Gamma \vdash M_1 : A \to B \qquad \Gamma \vdash M_2 : A \to B}{\Gamma \vdash \textbf{if}\ L\ \textbf{then}\ M_1\ \textbf{else}\ M_2 : A \to B} \qquad \Gamma \vdash N : A}{\Gamma \vdash (\textbf{if}\ L\ \textbf{then}\ M_1\ \textbf{else}\ M_2)\ N : B}$$

and can therefore show:

$$\dfrac{\Gamma \vdash L : \texttt{Bool} \qquad \dfrac{\Gamma \vdash M_1 : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M_1\ N : B} \qquad \dfrac{\Gamma \vdash M_2 : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M_2\ N : B}}{\Gamma \vdash \textbf{if}\ L\ \textbf{then}\ M_1\ N\ \textbf{else}\ M_2\ N : B}$$

- Type instantiation:
  - $(\Lambda\alpha.M)\ C \rightsquigarrow M[\alpha := C]$: follows from the constructor substitution lemma (Lemma 6.25).
  - $(\textbf{if}\ L\ \textbf{then}\ M_1\ \textbf{else}\ M_2)\ C \rightsquigarrow \textbf{if}\ L\ \textbf{then}\ M_1\ C\ \textbf{else}\ M_2\ C$: hoisting if-then-else out of the term works the same as application above.
- Fixpoint: follows from the substitution lemma (Lemma 6.25).
- 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.
  - $\textbf{for}\ (x \leftarrow [])\ N \rightsquigarrow []$: $[]$ has any list type and N has a list type.
  - $\textbf{for}\ (x \leftarrow [M])\ N \rightsquigarrow N[x := M]$: by substitution (Lemma 6.25).
  - $\textbf{for}\ (x \leftarrow M_1 +\!\!+ M_2)\ N \rightsquigarrow (\textbf{for}\ (x \leftarrow M_1)\ N) +\!\!+ (\textbf{for}\ (x \leftarrow M_2)\ N)$: reorder rules.
  - $\textbf{for}\ (x \leftarrow \textbf{for}\ (y \leftarrow L)\ M)\ N \rightsquigarrow \textbf{for}\ (y \leftarrow L)\ \textbf{for}\ (x \leftarrow M)\ N$:
    We have:

$$\dfrac{\dfrac{\Gamma \vdash L : [A_L] \qquad \Gamma, y : A_L \vdash M : [A_M]}{\Gamma \vdash \textbf{for}\ (y \leftarrow L)\ M : [A_M]} \qquad \Gamma, x : A_M \vdash N : [A_N]}{\Gamma \vdash \textbf{for}\ (x \leftarrow \textbf{for}\ (y \leftarrow L)\ M)\ N : [A_N]}$$

We need:

$$\frac{\Gamma \vdash L : [A_L] \qquad \dfrac{\Gamma, y : A_L \vdash M : [A_M] \qquad \Gamma, y : A_L, x : A_M \vdash N : [A_N]}{\Gamma, y : A_L \vdash \mathbf{for}\ (x \leftarrow M)\ N : [A_N]}}{\Gamma \vdash \mathbf{for}\ (y \leftarrow L)\ \mathbf{for}\ (x \leftarrow M)\ N : [A_N]}$$

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 6.26) and context swap (Lemma 6.27).

- 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:

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

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

- Case **rfold**: Typing rule:

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

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 $\mathtt{T}(C)$. M has type $\mathtt{T}(C)$. L has type $\mathtt{T}(C) \to \mathtt{T}(C) \to \mathtt{T}(C)$. Each $N.l_i$ has type $\mathtt{T}(C)$, because N has a record type obtained by mapping the constant function with result C over row S.

- Typecase typing rule:

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

Reduction rules:

- **typecase** $\texttt{Bool}^*$ **of** $(M_B, M_I, M_S, \beta.M_L, \rho.M_R, \gamma.M_T) \rightsquigarrow M_B$

  Need to show that $M_B : B[\alpha := \texttt{Bool}^*]$, which is one of our hypotheses.

- **typecase** $\texttt{List}^*\ C$ **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 $B[\alpha := \texttt{List}^*\ C]$, the same as the typing rule.

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

  Instantiating the constructor substitution lemma (Lemma 6.25) gives us

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

  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:

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

  which is the same as

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

The other cases are analogous.

- Case **tracecase**: Typing rule:

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

  Reductions:

  - **tracecase** 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. □

6.5.3 Links^{T} normal form

Unlike in typical progress and preservation proofs, our goal is not to reduce a program to a value like a number, string, or even a closure. Rather, we perform rewriting to partially evaluate parts of the program that are independent of values in the database. The goal is in particular to eliminate all occurrences of language constructs we cannot translate to Sql. The Links^{T} normal form is a crucial component of the argument for success. In combination with the progress lemma in the next section, it tells us what terms look like after exhaustive application of the normalization rules. Compared to Links^{T} source terms, the Links^{T} normal form shows that certain combinations of expressions are impossible after normalization, like an application of a literal function. Other terms necessarily remain, like iteration over a table (but not a literal list). We also see that some constructs are in normal form that are not in our target, the nested relational calculus. We will later argue that these cannot actually appear in queries, usually because queries are closed and have restricted types.

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 constr.	S	$::=$	$\cdot \mid l^P : C; S \mid U$
Neutral row constr.	U	$::=$	$\rho \mid l^P : C; U \mid \textbf{Rmap}\ C\ U$
Normal terms	M, N	$::=$	$F \mid c \mid \lambda x : A.M \mid \Lambda\alpha : K.M$
		$\mid$	$\textbf{if}\ H\ \textbf{then}\ M\ \textbf{else}\ N \mid M + N$
		$\mid$	$\langle\rangle \mid \langle l = M; N\rangle \mid \textbf{rmap}^U\ M\ N$
		$\mid$	$[] \mid [M] \mid M +\!\!+ N \mid \textbf{for}\ (x \leftarrow T)\ N \mid \textbf{table}\ n\ \langle R\rangle$
		$\mid$	$\texttt{Lit}\ M \mid \texttt{If}\ M \mid \texttt{For}\ C\ M \mid \texttt{Cell}\ M \mid \texttt{OpEq}\ C\ M \mid \texttt{OpPlus}\ M$
Neutral terms	F	$::=$	$x \mid P.l \mid F\ M \mid F\ C \mid \textbf{rfold}^U\ L\ M\ N$
		$\mid$	$\textbf{tracecase}\ F\ \textbf{of}\ (x.M_L, x.M_I, \alpha.x.M_F, x.M_C, \alpha.x.M_E, x.M_P)$
		$\mid$	$\textbf{typecase}\ E\ \textbf{of}\ (M_B, M_I, M_S, \beta.M_L, \rho.M_R, \beta.M_T)$
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 6.30: Links$^{\mathrm{T}}$ normal form.

The grammar of Links$^{\mathrm{T}}$ normal forms is given in Figure 6.30. Constructors C in Links$^{\mathrm{T}}$ normal form differ from constructors in non-normal Links$^{\mathrm{T}}$ in the application and **Typerec** forms. The idea is that the normal form only includes type applications and **Typerec** forms which are blocked from further reduction by an unbound type variable. For example, $(\lambda\alpha.\texttt{Bool}^*)\ \texttt{Int}^*$ is not in Links$^{\mathrm{T}}$ normal form (and we do not want it to be in normal form, because it can reduce to $\texttt{Bool}^*$). On the other hand, $\alpha\ \texttt{Int}^*$ is stuck and in normal form.

Remark 6.31. *Constructors E and 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 closed query terms in normal form. Such terms do not contain free type variables, so E and U collapse into nothing, and terms built from E and U (like **rmap**) cannot appear.

Neutral terms F are those stuck on a free variable x, a stuck constructor E, or a stuck row constructor U. We also restrict what can appear as the condition in an if-then-else, the iteratee of a list comprehension, and in the record position of a projection. We will later argue that in a query term, all variables are references

into tables and therefore everything that is not nested relational calculus will disappear. The reason for having separate syntactic sorts H, P, and T is to make this argument independent of types.

6.5.4 Progress

In this section we prove progress lemmas: a well-typed Links^{T} term is either in normal form or it reduces to some other term, and similarly for constructors and row constructors. The proofs are not surprising — all of the creative work is in the reduction relations and normal forms.

Lemma 6.32 (Constructor and row constructor progress)**.** *For all well-kinded* Links^{T} *type constructors C and row constructors S, we have that they are either in* Links^{T} *normal form (Figure 6.30), or there is a type constructor C' with $C \rightsquigarrow C'$, or row constructor S' with $S \rightsquigarrow S'$, respectively.*

Proof. By induction on the kinding derivation of C or S (see Figure 6.4).

- 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 $l^P : C; S$: by IH applied to C and S we have three cases:
  - If $C \rightsquigarrow C'$, then $l^P : C; S \rightsquigarrow l^P : C'; S$.
  - If $S \rightsquigarrow S'$, then $l^P : C; S \rightsquigarrow l^P : C; S'$.
  - If C and S are in normal form, then $l^P : C; S$ 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 $l^P : D; S'$: **Rmap** C $(l^P : D; S') \rightsquigarrow (l^P : C\ D; \textbf{Rmap}\ C\ S')$.
  - Case **Rmap** D U: **Rmap** C (**Rmap** D U) is in normal form.
  - Case ρ: **Rmap** C ρ is in normal form.
- The row variable ρ is in normal form. □

Lemma 6.33 (Progress)**.** *For all well-typed* Links$^{\mathrm{T}}$ *terms M, either M is in* Links$^{\mathrm{T}}$ *normal form* (*Figure 6.30*)*, or there is a* Links$^{\mathrm{T}}$ *term M' with $M \rightsquigarrow M'$.*

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^{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^{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^{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 6.32 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^{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 6.32 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^{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 6.32 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^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. □

6.5.5 Queries normalize to nested relational calculus

In this section, we further restrict the LINKS^T normal form for queries, that is, closed expressions of nested relational type. The general idea is, that because queries are closed and have nested relational type, they cannot possibly contain functions, polymorphism, record map, etc. The proof relies heavily on the exact definition of the normal form. In particular, the sorts U and E (Figure 6.30) have type and row variables as the only base cases. Queries in normal form do not contain any type or row variables and therefore do not contain any terms that are built using U or E, like **`rmap`** or **`typecase`**. We cannot rid ourselves of term variables — and thus terms they appear in — in quite the same manner, because **`for`**-comprehensions bind term variables and can appear in normal forms (and indeed must appear, otherwise we would have no need for generating database queries in the first place). However, since queries are closed, the only variables that appear are bound by comprehensions over tables and thus have closed record types with labels of base types. We capture this property in the definition of query contexts (Definition 6.34). We can use this to show a lemma which states that all neutral terms F are actually variables or projections of variables. We use this lemma to argue that queries are free of, for example, applications $F\ M$, because F collapses to x or $x.l$ and we know that all variables have table row types (closed records with labels of base type) and thus the application would be ill-typed. This argumentation extends to the other constructs that are in LINKS^T normal form but not in nested relational calculus, and therefore we

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

Figure 6.36: Target normal form for queries: NRC.

conclude that closed queries in normal form are in nested relational calculus.

Definition 6.34 (Query context)**.** A query context does not contain type or row variables and all term variables have closed record type with labels of base type.

- 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.

Lemma 6.35. *A term in* LINKS$^{\mathrm{T}}$ *normal form* (*Figure 6.30*) *that matches the grammar for F and 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, 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 $(\textbf{rmap}^U\ M\ N).l$. 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 $x.l$. □

Theorem 6.37. *If M is a term in* LINKS$^{\mathrm{T}}$ *normal form with a nested relational type in a query context* Γ*, then M is in the nested relational calculus* (*Figure 6.36*)*.*

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 6.35 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 6.35 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 6.35 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 6.31), 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 6.31), 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 6.35 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 6.35 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 6.31). In a query context which only binds term variables, this cannot possibly be well-typed — a contradiction. □

6.6 Implementation

We implemented a prototype of the LINKS^T self-tracing transformation and normalization procedure in HASKELL. We also implemented the `value`, `wherep`, and `lineage` trace analysis functions, as described in Section 6.3. This is just enough to report some initial results on provenance through trace analysis on a couple of examples.

Recall the boat tours query from Figure 2.4 on page 27. The query below is a minor variant (with reordered comprehensions and moved conditions) in LINKS^T.

```
for (x <- table "agencies" ⟨..⟩)
for (y <- table "externalTours" ⟨..⟩)
  if (x.name == y.name && y.type == "boat")
  then [⟨name = y.name, phone = x.phone⟩] else []
```

We can report that tracing the above query, applying the `value` trace analysis function, and normalizing again results in the exact same query.

More interestingly, we can trace the query and apply the `wherep` trace analysis function to extract where-provenance. The normalized query is shown below. The only difference to the query generated by LINKS^W is that we use a record of data, table, column, and row in LINKS^T, but a pair of data and provenance triple in LINKS^W.

```
for (x <- table "agencies" ⟨..⟩)
```

```
for (y <- table "externalTours" ⟨..⟩)
  if (x.name == y.name && y.type == "boat")
  then [⟨name = ⟨table = "externalTours", column = "name",
                 row = y.oid, data = y.name⟩,
         phone = ⟨table = "agencies", column = "phone",
                  row = x.oid, data = x.phone⟩⟩]
  else []
```

Doing the same with `lineage` once again shows the problem with tracing at the leaves and combining annotations using concatenation: Every row of the boat tours example has 15 lineage annotations, instead of 2, as seen below.

```
for (x <- table "agencies" ⟨..⟩)
for (y <- table "externalTours" ⟨..⟩)
  if (x.name == y.name && y.type == "boat")
  then [⟨data = ⟨name = y.name, phone = x.phone⟩,
         lineage = [⟨table = "agencies", row = x.oid⟩] ++
                   [⟨table = "agencies", row = x.oid⟩] ++
                   [⟨table = "agencies", row = x.oid⟩] ++
                   [⟨table = "externalTours", row = y.oid⟩] ++
                   [⟨table = "externalTours", row = y.oid⟩] ++
                   [⟨table = "externalTours", row = y.oid⟩] ++
                   [⟨table = "externalTours", row = y.oid⟩] ++
                   [⟨table = "externalTours", row = y.oid⟩] ++
                   [⟨table = "agencies", row = x.oid⟩] ++
                   [⟨table = "agencies", row = x.oid⟩] ++
                   [⟨table = "agencies", row = x.oid⟩] ++
                   [⟨table = "externalTours", row = y.oid⟩] ++
                   [⟨table = "externalTours", row = y.oid⟩] ++
                   [⟨table = "externalTours", row = y.oid⟩] ++
                   [⟨table = "externalTours", row = y.oid⟩] ++
                   [⟨table = "agencies", row = x.oid⟩]⟩]
  else []
```

We did not implement set semantics, or a special operator to remove static duplication. Something along those lines is clearly needed for provenance based on trace analysis.

Below is a variant of the example query from Chapter 4, after being traced, analyzed with the `value` trace analysis function, and normalized. Again, normalization removes all traces of tracing and results in a reasonable query.

```
for (x <- table "presidents" ⟨nth: Int, name: String⟩)
  [⟨dates = for (y <- table "inaugurations" ⟨..⟩)
```

```
            for (z <- table "metro" ⟨..⟩)
              if (y.nth == x.nth && z.date == y.date &&
                  z.time == 11 && z.trips >= 193000)
              then [y.date] else [],
    name = x.name⟩]
```

Where-provenance of nested results works, as seen below.

```
for (x <- table "presidents" ⟨nth: Int, name: String⟩)
  [⟨dates = for (y <- table "inaugurations" ⟨..⟩)
            for (z <- table "metro" ⟨..⟩)
              if (y.nth == x.nth && z.date == y.date &&
                  z.time == 11 && z.trips >= 193000)
              then [⟨table = "inaugurations", column = "date",
                     row = y.oid, data = y.date⟩]
              else [],
    name = ⟨table = "presidents", column = "name",
            row = x.oid, data = x.name⟩⟩]
```

Figure 6.38 shows the normalized query after tracing and applying the `lineage` trace analysis function. This time, we removed the obviously duplicated lineage annotations. Still, the last line adds the presidents table to the lineage of the outer collection. This is one instance where static deduplication in set union is not entirely trivial. We would have to push the singleton list with the presidents annotation into the nested comprehensions and into the else-branch (the then-branch stays as it is) to entirely remove duplicates.

We also tried some of the benchmark queries from Chapter 5. Other than the duplication of annotations in the lineage queries, the generated code looks reasonable. Normalization performance is good enough to be unnoticeable, even with a naive implementation of the reduction rules that only performs a single step at a time and re-traverses the expression until it finds the next redex in every step.

```
for (x <- table "presidents" ⟨nth: Int, name: String⟩)
  [⟨data = ⟨dates = for (y <- table "inaugurations" ⟨..⟩)
                    for (z <- table "metro" ⟨..⟩)
                      if (y.nth == x.nth && z.date == y.date &&
                          z.time == 11 && z.trips >= 193000)
                      then [⟨data = y.date,
                             lineage = [⟨table = "presidents",
                                        row = x.oid⟩,
                                       ⟨table = "inaugurations",
                                        row = y.oid⟩,
                                       ⟨table = "metro",
                                        row = z.oid⟩]⟩]
                      else [],
            name = x.name⟩,
    lineage = (for (y <- table "inaugurations" ⟨..⟩)
               for (z <- table "metro" ⟨..⟩)
                 if (y.nth == x.nth && z.date == y.date &&
                     z.time == 11 && z.trips >= 193000)
                 then [⟨table = "presidents", row = x.oid⟩,
                       ⟨table = "inaugurations", row = y.oid⟩,
                       ⟨table = "metro", row = z.oid⟩]
                 else []) ++
              [⟨table = "presidents", row = x.oid⟩]⟩]
```

Figure 6.38: Trace analysis with `lineage` of the example query from Chapter 4 with obvious lineage annotation duplication manually removed.

6.7 Related and future work

We did not prove strong normalization and in the presence of **fix** we, of course, do not expect strong normalization to hold. However, it would be nice to have some guarantee about the termination behavior of normalization like: if call-by-name evaluation would terminate then normalization terminates. An implementation could use this to guarantee termination of trace analysis functions by, for example, enforcing structural recursion.

Extracting provenance from traces is not a new idea [Acar et al., 2012; Cheney et al., 2014a; Müller et al., 2018]. What makes our work different is that trace analyses are defined in the language itself. In combination with query normalization, this makes LINKS^T the first, to our knowledge, system that can execute user-defined query trace analysis inside the database system.

Compared to some of the other work on tracing, 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, for example, functions in the original query code. Tree-shaped traces with explicit representation of variables like those proposed by Cheney et al. [2014a] in particular, seem to make writing well-typed provenance extraction functions more difficult. It can be done with dependent types — we prototyped such traces and trace analysis functions in IDRIS— but this is not necessarily the direction we expect LINKS to go. Müller et al. [2018] only record dynamic control flow decisions in traces and interpret the static query together with the dynamic trace to recover provenance. It would be interesting to see whether we could do something similar in a more language-integrated way.

We propose to use query shredding [Cheney et al., 2014c] to turn nested relational calculus into a bounded number of SQL queries because it is implemented in LINKS already. Alternatively, we could use other query compilation strategies like that employed by DSH [Ulrich and Grust, 2015] which is based on the flattening transformation [Blelloch and Sabot, 1990] or even target other execution backends by translating nested relational calculus further using, for example, the query compiler Q*CERT [Auerbach et al., 2017].

Links^T follows Links tradition and uses a normalization procedure that eliminates language constructs with no obvious counterpart in Sql, rather than attempt to encode these constructs. However, the latter might be an option too. Crary et al. [2002] introduce λ_r, a variant of λ_i^{ML} where runtime type information is explicitly represented as a datatype when necessary. Giorgidze et al. [2013] show how to translate algebraic datatypes to Sql and Grust and Ulrich [2013] translate functions to Sql. It seems plausible that one could put these together to translate all of Links^T to Sql without necessarily normalizing to eliminate functions, traces, and typecase. Indeed, the restriction to query types would become unnecessary. One might still want to do some normalization, or partial evaluation, to avoid computing large traces where possible.

Links^T builds on λ_i^{ML} [Morrisett, 1995]. λ_r 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 Crary et al. [2002]. An implementation would benefit from the latter. The former points do not matter much in the context of Links, because it is interpreted anyway, which makes it easy to access to type information at runtime, and we expect normalization time to typically be irrelevant compared to the time it takes to open a database connection, let alone execute a query.

Links^T features generic record programming in the form of record mapping and folding. These were the simplest extensions to how Links currently implements records and row types [Lindley and Cheney, 2012] we could come up with, that would meet the requirements of trace analysis. Other languages with generic record programming constructs include Ur/Web and PureScript.

Ur/Web [Chlipala, 2015] features "first class, type-level names and records" [Chlipala, 2010]. Its generic and metaprogramming features seem suitable for our needs. At the moment it lacks a powerful query compilation strategy, however. Type inference for Links^T is an open problem. Type inference for Ur/Web is undecidable. However, Chlipala [2010] 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^T, too.

PureScript is a Haskell-like language with first class records based on row types. It supports generic record programming by converting back and forth between rows and type-level lists, and general type-level computation with multi-parameter type classes, functional dependencies, and parts of instance

chains [Morris and Jones, 2010]. This is enough to encode type-level functions like `WHERE`, but not a particularly direct or pleasant way to program. Links does not at the moment have anything like type classes. It is not clear that implementing all of this machinery just to avoid record map and fold is cost-effective. However, type classes are widely useful in general and we could also use them to restrict queries to query types instead of having this built into the typechecker, so it might be worthwhile.

Haskell's support for records is syntactic sugar for algebraic datatypes with accessor functions and does not currently extend to record map and fold. Even so, it is almost certainly possible to encode those, as well as typecase and the necessary type-level computations in Haskell. It would be very interesting to see an implementation of Links^T on top of Dsh [Giorgidze et al., 2011; Stolarek and Cheney, 2018; Ulrich and Grust, 2015].

As mentioned before, we have designed Links^T to be as easy an extension of Links as possible, but it is not implemented yet. This should probably be the next step towards flexible, efficient, and effective language-integrated provenance.

Chapter 7

Conclusions

This dissertation introduces language-integrated provenance — the idea that a programming language can be a provenance system in its own right, capable of answering data provenance questions about queries, without any special support for provenance from the underlying database system. Language-integrated query is the key enabling technology for encoding provenance annotations and to turn even user-defined provenance propagation behavior into efficient SQL queries to be executed by mainstream database systems.

There are some shortcomings, open questions, and opportunities for interesting future work. Programs, in particular web servers, may run much longer than the duration of a database query or even a transaction. It would be interesting to extend the correctness properties for where-provenance and lineage to say something about the meaning of provenance annotations in the presence of database updates.

LINKS lacks some features that are commonly found in other query languages, such as grouping and aggregations. It would be interesting to add support for them to LINKS and its provenance variants, or to further explore language-integrated provenance in a language that already supports them. LINKS$^{\mathrm{T}}$ in particular would benefit from support for mixed set, multiset, and list semantics in a single query. Features not commonly found in query languages are algebraic datatypes and first-class functions, which can nevertheless be translated to SQL [Giorgidze et al., 2013; Grust and Ulrich, 2013]. These are particularly interesting for language-integrated provenance because they might make the implementation of LINKS$^{\mathrm{T}}$ easier and they could narrow the gap between provenance of just database queries and provenance of more general-purpose language constructs.

There are challenges and opportunities in implementing Links^{T}, specifically. Writing a trace analysis function that returns both where-provenance and lineage should be straightforward. However, it would be more satisfying to find a general way to structure analysis functions such that any number of different forms of provenance can be combined. While we have proven basic provenance correctness properties for Links^{W} and Links^{L}, we have no such proofs for the respective trace analysis functions, so Links^{T} falls short of a strict interpretation of Requirement 1. Other well-known forms of provenance may or may not be expressible as trace analysis functions at all, given the power of the language and normalization algorithm and the structure and information content of traces. Their implementation might require changes to the structure and content of traces, the language of trace analysis functions, and its normalization algorithm.

Despite the limitations and remaining open questions, this dissertation shows how to solve the problems set out in the introduction. Looking back at the requirements for provenance systems set out by Glavic, Miller, and Alonso [2013], we claim that language-integrated provenance has

1. "Support for different types of provenance with sound semantics."

   Links^{W} and Links^{L} demonstrate language support for one form of provenance each. We prove basic provenance correctness properties for both of them. Links^{W} in particular is upfront about what can and cannot have where-provenance and rejects queries that ask for the impossible.

   Links^{T} goes further than most previous provenance systems and allows users to define their own forms of provenance by writing functions to extract information from the trace of a query execution.

2. "Support for provenance generation for complex [queries]."

   The compositional nature of the query translations combined with query normalization means that provenance generation works on all queries, if it is defined in the first place. Links supports first-class functions and nested results, which makes it richer than many other query languages. However, there is still work to do to extend this to grouping and aggregations.

3. "Support for complex queries over provenance information."

   Provenance information uses the same nested relational data model as other query data. Thanks, again, to compositional query transformations

and query normalization, the whole language can be used in queries over provenance information.

4. "Support for large databases."

   Provenance is computed on demand and not stored in the database. Query normalization produces reasonably efficient queries using standard SQL features that are easy for query planners to optimize.

   In Chapter 5, we evaluate the performance overhead for querying where-provenance and lineage in LINKS^{W} and LINKS^{L} and conclude that it is comparable to existing research prototypes of specialized provenance systems. This has been partially confirmed by Lee, Ludäscher, and Glavic [2018], who have compared the performance of LINKS^{L} to their provenance system PUG and found them competitive.

Beyond the basic requirements for provenance systems, the provenance variants of LINKS provide different models for interacting safely with provenance data in a client program. LINKS^{T} exceeds our initial goals and offers programmable provenance support that allows users to define their own forms of provenance, rather than just a choice between a number of built-in options.

Language-integrated provenance also offers independence from special support for provenance from the underlying database system. While we have personally only used LINKS together with POSTGRESQL so far, the generated queries use standard SQL features and should be portable to most mainstream database systems.

Rewriting all programs in a research programming language to be able to query provenance may seem even less appealing than switching to a research prototype of a database system. Fortunately, the general idea of language-integrated provenance is applicable to other languages with sufficiently powerful language-integrated queries. Stolarek and Cheney [2018] have already implemented the translations employed by LINKS^{W} and LINKS^{L} in HASKELL, using the DSH library [Ulrich and Grust, 2015]. A similar embedding could work for OCAML, for which there is a normalization-based query library called QUEΛ [Suzuki et al., 2016]. IDRIS [Brady, 2013] and other dependently-typed languages might be powerful enough to implement even LINKS^{T} as a library. Languages that provide type safety for language-integrated queries but no normalization, such as C#, F#, and UR/WEB [Chlipala, 2015], would require much more careful provenance

translations to still produce reasonable queries. Languages without support for nested collections in queries would require a different representation of lineage. It should be possible to find syntax-directed variations of the type-directed provenance translations for use in untyped languages. Those would of course not provide the same type safety guarantees, but then their users supposedly have found ways to cope with that.

For the future, we hope that language-integration will make provenance more widely available. This should help programmers write robust, provenance-aware programs; make computation more transparent; answer questions about the reliability and sources of data; and generally be a small step towards making the dreams of Cheney, Chong, Foster, Seltzer, and Vansummeren [2009b], myself, and other provenance researchers come true.

Bibliography

Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A core calculus for provenance. In *Principles of Security and Trust*, pages 410–429. Springer, 2012. ISBN 978-3-642-28641-4.

Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB 2006, pages 1151–1154. VLDB Endowment, 2006. URL `https://dl.acm.org/citation.cfm?id=1182635.1164231`.

Yael Amsterdamer, Daniel Deutch, and Val Tannen. On the limitations of provenance for queries with difference. In *3rd Workshop on the Theory and Practice of Provenance*, TaPP 2011, 2011a. URL `https://www.usenix.org/conference/tapp11/limitations-provenance-queries-difference`.

Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS 2011, pages 153–164. ACM, 2011b. ISBN 978-1-4503-0660-7. doi: 10.1145/1989284.1989302. URL `http://doi.acm.org/10.1145/1989284.1989302`.

Bahareh Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. A generic provenance middleware for queries, updates, and transactions. In *6th USENIX Workshop on the Theory and Practice of Provenance*, TaPP 2014. USENIX Association, 2014. URL `https://www.usenix.org/conference/tapp2014/agenda/presentation/arab`.

Michael Ashburner, Catherine A. Ball, Judith A. Blake, David Botstein, Heather Butler, J. Michael Cherry, Allan P. Davis, Kara Dolinski, Selina S. Dwight, Janan T. Eppig, Midori A. Harris, David P. Hill, Laurie Issel-Tarver, Andrew Kasarskis, Suzanna Lewis, John C. Matese, Joel E. Richardson, Martin Ringwald, Gerald M. Rubin, and Gavin Sherlock. Gene Ontology: tool for the unification of biology. *Nature Genetics*, 25(1):25–29, May 2000. ISSN 1061-4036. doi: 10.1038/75556. URL `https://dx.doi.org/10.1038/75556`.

Joshua S. Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. Prototyping a query compiler using Coq (experience report). *Proceedings of the ACM on Programming Languages*, 1(ICFP):9:1–9:15, August 2017.

ISSN 2475-1421. doi: 10.1145/3110253. URL https://doi.acm.org/10.1145/3110253.

Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, Martin Theobald, and Jennifer Widom. Databases with uncertainty and lineage. *The VLDB Journal*, 17(2):243–264, Mar 2008. doi: 10.1007/s00778-007-0080-z. URL https://dx.doi.org/10.1007/s00778-007-0080-z.

Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. *The VLDB Journal*, 14(4):373–396, 2005. doi: 10.1007/s00778-005-0156-6. URL https://dx.doi.org/10.1007/s00778-005-0156-6.

Guy Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8: 119–134, 1990.

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5): 552–593, 2013. doi: 10.1017/S095679681300018X.

Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comp. Sci.*, 149(1):3–48, 1995.

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Data provenance: Some basic issues. In *Proceedings of FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *LNCS*, pages 87 – 93. Springer, Dec 2000. ISBN 978-3-540-44450-3. doi: 10.1007/3-540-44450-5_6. URL https://dx.doi.org/10.1007/3-540-44450-5_6. Invited Paper.

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *ICDT 2001*, number 1973 in LNCS, pages 316–330. Springer, 2001. ISBN 978-3-540-41456-8. doi: 10.1007/3-540-44503-X_20. URL https://dx.doi.org/10.1007/3-540-44503-X_20.

Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems*, 33(4):28:1–28:47, December 2008. ISSN 0362-5915. doi: 10.1145/1412331.1412340. URL https://doi.acm.org/10.1145/1412331.1412340.

James Cheney. Program slicing and data provenance. *IEEE Data Engineering Bulletin*, 30(4):22–28, 2007. URL http://sites.computer.org/debull/A07dec/cheney.pdf.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, April

2009a. ISSN 1931-7883. doi: 10.1561/1900000006. URL https://dx.doi.org/10.1561/1900000006.

James Cheney, Stephen Chong, Nate Foster, Margo Seltzer, and Stijn Vansummeren. Provenance: A future history. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA 2009, pages 957–964. ACM, 2009b. ISBN 978-1-60558-768-4. doi: 10.1145/1639950.1640064. URL http://doi.acm.org/10.1145/1639950.1640064.

James Cheney, Amal Ahmed, and Umut A. Acar. Provenance as dependency analysis. *Mathematical Structures in Computer Science*, 21:1301–1337, 12 2011. ISSN 1469-8072. doi: 10.1017/S0960129511000211. URL https://journals.cambridge.org/article_S0960129511000211.

James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2013, pages 403–416. ACM, 2013. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500586. URL https://doi.acm.org/10.1145/2500365.2500586.

James Cheney, Amal Ahmed, and Umut A. Acar. Database queries that explain their work. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, PPDP 2014, pages 271–282. ACM, 2014a. ISBN 978-1-4503-2947-7. doi: 10.1145/2643135.2643143. URL https://doi.acm.org/10.1145/2643135.2643143.

James Cheney, Sam Lindley, Gabriel Radanne, and Philip Wadler. Effective quotation: Relating approaches to language-integrated query. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, pages 15–26. ACM, 2014b. ISBN 978-1-4503-2619-3. doi: 10.1145/2543728.2543738. URL https://doi.acm.org/10.1145/2543728.2543738.

James Cheney, Sam Lindley, and Philip Wadler. 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, pages 1027–1038. ACM, 2014c. ISBN 978-1-4503-2376-5. doi: 10.1145/2588555.2612186. URL https://doi.acm.org/10.1145/2588555.2612186.

Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. DBNotes: a Post-It system for relational databases based on provenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 942–944, 2005. doi: 10.1145/1066157.1066296. URL https://doi.acm.org/10.1145/1066157.1066296.

Adam Chlipala. 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, pages 122–133. ACM, 2010. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806612. URL https://doi.acm.org/10.1145/1806596.1806612.

Adam Chlipala. 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, pages 153–165. ACM, 2015. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677004. URL https://doi.acm.org/10.1145/2676726.2677004.

David Coleman. Presidential inauguration dates, Washington to Trump, 2017. URL https://historyinpieces.com/research/presidential-inauguration-dates. Accessed 30 October 2018.

Ezra Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL 2009*, volume 5708 of *LNCS*, pages 36–51. Springer, 2009. ISBN 978-3-642-03792-4. doi: 10.1007/978-3-642-03793-1_3. URL https://dx.doi.org/10.1007/978-3-642-03793-1_3.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, FMCO 2006, pages 266–296. Springer, 2007. ISBN 3-540-74791-5, 978-3-540-74791-8. URL https://dl.acm.org/citation.cfm?id=1777707.1777724.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 205–220, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89329-5. doi: 10.1007/978-3-540-89330-1_15. URL http://dx.doi.org/10.1007/978-3-540-89330-1_15.

Brian J. Corcoran, Nikhil Swamy, and Michael W. Hicks. Cross-tier, label-based security enforcement for web applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2009, pages 269–282, 2009. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559875. URL https://doi.acm.org/10.1145/1559845.1559875.

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

Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *Proceedings of 16th International Conference on Data Engineering*, pages 367–378, Feb 2000a. doi: 10.1109/ICDE.2000.839437. URL https://dx.doi.org/10.1109/ICDE.2000.839437.

Yingwei Cui and Jennifer Widom. Storing auxiliary data for efficient maintenance and lineage tracing of complex views. *Proceedings of the 2nd International Workshop on Design and Management of Data Warehouses*, June 2000b.

Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*,

25(2):179–227, June 2000. ISSN 0362-5915. doi: 10.1145/357775.357777. URL https://doi.acm.org/10.1145/357775.357777.

Stefan Fehrenbach and James Cheney. Language-integrated provenance in Links. In *7th USENIX Workshop on the Theory and Practice of Provenance*, TaPP 2015. USENIX Association, July 2015. URL https://www.usenix.org/conference/tapp15/workshop-program/presentation/fehrenbach.

Stefan Fehrenbach and James Cheney. Language-integrated provenance. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP 2016, pages 214–227. ACM, 2016. ISBN 978-1-4503-4148-6. doi: 10.1145/2967973.2968604. URL http://doi.acm.org/10.1145/2967973.2968604.

Stefan Fehrenbach and James Cheney. Language-integrated provenance. *Science of Computer Programming*, 155:103 – 145, 2018. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2017.08.009. URL https://www.sciencedirect.com/science/article/pii/S0167642317301685. Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2016.

Stefan Fehrenbach and James Cheney. Language-integrated provenance by trace analysis. *DBPL 2019*, 2019. To appear.

Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979. doi: 10.1002/spe.4380090402. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090402.

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: Queries and provenance. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS 2008, pages 271–280. ACM, 2008. ISBN 978-1-60558-152-1. doi: 10.1145/1376916.1376954. URL http://doi.acm.org/10.1145/1376916.1376954.

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: Session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, January 2019. ISSN 2475-1421. doi: 10.1145/3290341. URL http://doi.acm.org/10.1145/3290341.

Wolfgang Gatterbauer, Alexandra Meliou, and Dan Suciu. Default-all is Dangerous! In *3rd USENIX Workshop on the Theory and Practice of Provenance*, TaPP 2011, Heraklion, Greece, June 2011.

George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell boards the ferry: Database-supported program execution for Haskell. In *IFL 2010*, pages 1–18. Springer, 2011. ISBN 978-3-642-24275-5. URL https://dl.acm.org/citation.cfm?id=2050135.2050136.

George Giorgidze, Torsten Grust, Alexander Ulrich, and Jeroen Weijers. Algebraic data types for language-integrated queries. In *Proceedings of the 2013 Workshop on Data Driven Functional Programming*, DDFP 2013, pages 5–10. ACM, 2013. ISBN 978-1-4503-1871-6. doi: 10.1145/2429376.2429379. URL https://doi.acm.org/10.1145/2429376.2429379.

Boris Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, University of Zürich, 2010. URL https://cs.iit.edu/%7edbgroup/pdfpubls/G10a.pdf.

Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE)*, pages 174–185, 2009. doi: 10.1109/ICDE.2009.15. URL https://dx.doi.org/10.1109/ICDE.2009.15.

Boris Glavic, Renée J. Miller, and Gustavo Alonso. Using SQL for efficient generation and querying of provenance information. In *In Search of Elegance in the Theory and Practice of Computation: Essays Dedicated to Peter Buneman*, volume 8000 of *LNCS*, pages 291–320. Springer, 2013. ISBN 978-3-642-41660-6. doi: 10.1007/978-3-642-41660-6_16. URL https://doi.org/10.1007/978-3-642-41660-6_16.

Todd J. Green. Containment of conjunctive queries on annotated relations. *Theory of Computing Systems*, 49(2):429–459, Aug 2011. ISSN 1433-0490. doi: 10.1007/s00224-011-9327-6. URL https://doi.org/10.1007/s00224-011-9327-6.

Todd J. Green and Val Tannen. The semiring framework for database provenance. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS 2017, pages 93–99. ACM, 2017. ISBN 978-1-4503-4198-1. doi: 10.1145/3034786.3056125. URL http://doi.acm.org/10.1145/3034786.3056125.

Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB 2007, pages 675–686. VLDB Endowment, 2007a. ISBN 978-1-59593-649-3. URL http://dl.acm.org/citation.cfm?id=1325851.1325929.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 31–40. ACM, 2007b. ISBN 978-1-59593-685-1. doi: 10.1145/1265530.1265535. URL https://doi.acm.org/10.1145/1265530.1265535.

Torsten Grust and Jan Rittinger. Observing SQL queries in their natural habitat. *ACM Transactions on Database Systems*, 38(1):3:1–3:33, April 2013. ISSN 0362-5915. doi: 10.1145/2445583.2445586. URL https://doi.acm.org/10.1145/2445583.2445586.

Torsten Grust and Alexander Ulrich. First-class functions for first-order database engines. In *Proceedings of the 14th International Symposium on Database Programming Languages*, 2013. URL https://arxiv.org/abs/1308.0158.

Torsten Grust, Maurice Van Keulen, and Jens Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29(1): 91–131, March 2004. ISSN 0362-5915. doi: 10.1145/974750.974754. URL http://doi.acm.org/10.1145/974750.974754.

Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-safe linq compilation. *Proceedings of the VLDB Endowment*, 3(1-2):162–172, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920866. URL http://dx.doi.org/10.14778/1920841.1920866.

Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 1995, pages 130–141. ACM, 1995. ISBN 0-89791-692-1. doi: 10.1145/199448.199475. URL https://doi.acm.org/10.1145/199448.199475.

Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26(6): 881–906, December 2017. ISSN 1066-8888. doi: 10.1007/s00778-017-0486-1. URL https://doi.org/10.1007/s00778-017-0486-1.

Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pages 15–27, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4435-7. doi: 10.1145/2976022.2976033. URL http://doi.acm.org/10.1145/2976022.2976033.

Rudi Horn, Roly Perera, and James Cheney. Incremental relational lenses. *Proceedings of the ACM on Programming Languages*, 2(ICFP):74:1–74:30, July 2018. ISSN 2475-1421. doi: 10.1145/3236769. URL http://doi.acm.org/10.1145/3236769.

Grigoris Karvounarakis and Todd J. Green. Semiring-annotated data: Queries and provenance? *SIGMOD Rec.*, 41(3):5–14, October 2012. ISSN 0163-5808. doi: 10.1145/2380776.2380778. URL http://doi.acm.org/10.1145/2380776.2380778.

Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2010, pages 951–962. ACM, 2010. ISBN

978-1-4503-0032-2. doi: 10.1145/1807167.1807269. URL http://doi.acm.org/10.1145/1807167.1807269.

Sven Köhler, Bertram Ludäscher, and Daniel Zinn. First-order provenance games. In *In Search of Elegance in the Theory and Practice of Computation: Essays Dedicated to Peter Buneman*, volume 8000 of *LNCS*, pages 382–399. Springer, 2013. ISBN 978-3-642-41660-6. doi: 10.1007/978-3-642-41660-6_20. URL https://doi.org/10.1007/978-3-642-41660-6_20.

Seokki Lee, Bertram Ludäscher, and Boris Glavic. Pug: a framework and practical implementation for why and why-not provenance. *The VLDB Journal*, Aug 2018. ISSN 0949-877X. doi: 10.1007/s00778-018-0518-5. URL https://doi.org/10.1007/s00778-018-0518-5.

Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI 2012, pages 91–102. ACM, 2012. ISBN 978-1-4503-1120-5. doi: 10.1145/2103786.2103798. URL https://doi.acm.org/10.1145/2103786.2103798.

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

Erik Meijer, Brian Beckman, and Gavin Bierman. 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, pages 706–706. ACM, 2006. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142552. URL https://doi.acm.org/10.1145/1142473.1142552.

J. Garrett Morris and Mark P. Jones. Instance chains: Type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2010, pages 375–386. ACM, 2010. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863596. URL https://doi.acm.org/10.1145/1863543.1863596.

Greg Morrisett. *Compiling with types*. PhD thesis, Carnegie Mellon University, 1995. URL https://www.cs.cmu.edu/~rwh/theses/morrisett.pdf.

Tobias Müller, Benjamin Dietrich, and Torsten Grust. You say 'what', I hear 'where' and 'why': (mis-)interpreting SQL to derive fine-grained provenance. *Proceedings of the VLDB Endowment*, 11(11):1536–1549, July 2018. ISSN 2150-8097. doi: 10.14778/3236187.3236204. URL https://doi.org/10.14778/3236187.3236204.

Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy, and Venkatesh Radhakrishnan. Provenance-aware query

optimization. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 473–484, April 2017. doi: 10.1109/ICDE.2017.104. URL https://dx.doi.org/10.1109/ICDE.2017.104.

Atsushi Ohori and Katsuhiro Ueno. Making Standard ML a practical database programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2011, pages 307–319. ACM, 2011. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034815. URL https://doi.acm.org/10.1145/2034773.2034815.

Oxford Dictionary. Definition: provenance (noun), 2018. URL https://en.oxforddictionaries.com/definition/provenance. Accessed 18 June 2018.

Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2012, pages 365–376. ACM, 2012. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364579. URL https://doi.acm.org/10.1145/2364527.2364579.

Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. Imperative functional programs that explain their work. *Proceedings of the ACM on Programming Languages*, 1(ICFP):14:1–14:28, August 2017. ISSN 2475-1421. doi: 10.1145/3110258. URL https://doi.acm.org/10.1145/3110258.

Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. ProvSQL: Provenance and probability management in PostgreSQL. *Proceedings of the VLDB Endowment*, 11(12):2034–2037, August 2018. ISSN 2150-8097. doi: 10.14778/3229863.3236253. URL https://doi.org/10.14778/3229863.3236253.

Lwin Khin Shar and Hee Beng Kuan Tan. Defeating SQL injection. *IEEE Computer*, 46(3):69–77, 2013. doi: 10.1109/MC.2012.283. URL https://dx.doi.org/10.1109/MC.2012.283.

Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *ACM SIGMOD Record*, 34(3):31–36, September 2005. ISSN 0163-5808. doi: 10.1145/1084805.1084812. URL http://doi.acm.org/10.1145/1084805.1084812.

Jan Stolarek and James Cheney. Language-integrated provenance in Haskell. *The Art, Science, and Engineering of Programming*, 2(3), 4 2018. ISSN 2473-7321. doi: 10.22152/programming-journal.org/2018/2/11. URL https://dx.doi.org/10.22152/programming-journal.org/2018/2/11.

Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. Finally, safely-extensible and efficient language-integrated query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2016, pages 37–48. ACM, 2016. ISBN 978-1-4503-4097-7. doi: 10.1145/2847538.2847542. URL http://doi.acm.org/10.1145/2847538.2847542.

Don Syme. Leveraging .NET meta-programming components from F#: Integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 Workshop on ML*, ML 2006, pages 43–54. ACM, 2006. ISBN 1-59593-483-9. doi: 10.1145/1159876.1159884. URL http://doi.acm.org/10.1145/1159876.1159884.

Wang-Chiew Tan. Research problems in data provenance. *IEEE Data Engineering Bulletin*, 27:45–52, 2004.

Yu Shyang Tan, Ryan K.L. Ko, and Geoff Holmes. Security and data accountability in distributed systems: A provenance survey. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1571–1578, Nov 2013. doi: 10.1109/HPCC.and.EUC.2013.221. URL https://dx.doi.org/10.1109/HPCC.and.EUC.2013.221.

The Gene Ontology Consortium. Expansion of the gene ontology knowledgebase and resources. *Nucleic Acids Research*, 45(D1):D331–D338, 2017. doi: 10.1093/nar/gkw1108. URL https://dx.doi.org/10.1093/nar/gkw1108.

Alexander Ulrich. A Ferry-based query backend for the Links programming language. Master's thesis, Eberhard Karls Universität Tübingen, Germany, March 2011. Diplomarbeit.

Alexander Ulrich and Torsten Grust. 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, pages 1421–1426. ACM, 2015. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2735359. URL https://doi.acm.org/10.1145/2723372.2735359.

Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359. ACM, 1989.

Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research* (*CIDR*), January 2005. URL https://dblp.uni-trier.de/db/conf/cidr/cidr2005.html#Widom05.

Janet L. Wiener, Himanshu Gupta, Wilburt J. Labio, Yue Zhuge, Hector Garcia-Molina, and Jennifer Widom. A system prototype for warehouse view maintenance. In *Post-Sigmod Workshop on Materialized Views*, 1995. URL http://ilpubs.stanford.edu:8090/109/.

Limsoon Wong. Normal forms and conservative extension properties for query languages over collection types. *Journal of Computer and System Sciences*, 52(3): 495 – 505, 1996. ISSN 0022-0000. doi: 10.1006/jcss.1996.0037. URL https://doi.org/10.1006/jcss.1996.0037.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10. USENIX Association, 2010. URL `https://dl.acm.org/citation.cfm?id=1863103.1863113`.