

## THE UNIVERSITY of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e. g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.


# Language Integrated Relational Lenses 

Rudi Horn



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


#### Abstract

Relational databases are ubiquitous. Such monolithic databases accumulate large amounts of data, yet applications typically only work on small portions of the data at a time. A subset of the database defined as a computation on the underlying tables is called a view. Querying views is helpful, but it is also desirable to update them and have these changes be applied to the underlying database. This view update problem has been the subject of much previous work before, but support by database servers is limited and only rarely available.

Lenses are a popular approach to bidirectional transformations, a generalization of the view update problem in databases to arbitrary data. However, perhaps surprisingly, lenses have seldom actually been used to implement updatable views in databases. Bohannon, Pierce and Vaughan [12] propose an approach to updatable views called relational lenses. However, to the best of our knowledge this proposal has not been implemented or evaluated prior to the work reported in this thesis.

This thesis proposes programming language support for relational lenses. Language integrated relational lenses support expressive and efficient view updates, without relying on updatable view support from the database server. By integrating relational lenses into the programming language, application development becomes easier and less error-prone, avoiding the impedance mismatch of having two programming languages. Integrating relational lenses into the language poses additional challenges. As defined by Bohannon et al. relational lenses completely recompute the database, making them inefficient as the database scales. The other challenge is that some parts of the well-formedness conditions are too general for implementation. Bohannon et al. specify predicates using possibly infinite abstract sets and define the type checking rules using relational algebra.

Incremental relational lenses equip relational lenses with change-propagating semantics that map small changes to the view into (potentially) small changes to the source tables. We prove that our incremental semantics are functionally equivalent to the non-incremental semantics, and our experimental results show orders of magnitude improvement over the non-incremental approach. This thesis introduces a concrete predicate syntax and shows how the required checks are performed on these predicates and show that they satisfy the abstract predi-


cate specifications. We discuss trade-offs between static predicates that are fully known at compile time vs dynamic predicates that are only known during execution and introduce hybrid predicates taking inspiration from both approaches. This thesis adapts the typing rules for relational lenses from sequential composition to a functional style of sub-expressions. We prove that any well-typed functional relational lens expression can derive a well-typed sequential lens.

We use these additions to relational lenses as the foundation for two practical implementations: an extension of the Links functional language and a library written in Haskell. The second implementation demonstrates how type-level computation can be used to implement relational lenses without changes to the compiler. These two implementations attest to the possibility of turning relational lenses into a practical language feature.

## Lay Summary

Many computer programs are required to persistently store data to function as intended. Applications can store arbitrary data in files, but storing large amounts of data so that they can be accessed efficiently is challenging. Rather than reinventing the wheel, applications can use existing software, called relational database management systems, to efficiently store data in tabular form. Data is fetched by the application by issuing queries, expressions that specify which data should be returned to the application.

The resulting subset of the database computed by a query is called a view. A desirable task is to not only compute such a view, but to also make changes to it and apply those changes to the database. The view update problem has been the subject of work before, but support for updatable views is only limited and rarely available.

In programming languages, the view-update problem is generalized to arbitrary data in the form of lenses. A lens is a combination of two functions. The first function computes the view from the data source, while the other function uses a changed view to update the source. A lens is well-behaved if it satisfies roundtripping guarantees. Lenses can be combined with other lenses to produce more complicated lenses. Bohannon, Pierce and Vaughan [12] propose the use of relational lenses for querying and updating data stored in relational tables. Relational lenses are well-behaved, making their behaviour predictable. However, prior to the work on this thesis, relational lenses were missing an implementation and have only been considered in isolated contexts.

We propose the integration of relational lenses into the programming language the application is written in. This makes it easy for the programmer to define and use relational lenses within the application. We show how the performance of updates to relational lens views can be improved as the database size increases, by only computing the changes to the database rather than recomputing the entire database. We present a formulation that is more suitable for the integration into programming languages, and show how to check if it is safe to construct a lens in this context. We demonstrate our work on implementations of relational lenses in two different programming languages. The implementations interact with existing relational database management systems.

## Acknowledgements

First and foremost, I would like to thank my supervisor James Cheney for the continuous guidance throughout both my second MSc as well as my PhD studies. I am grateful to James for his inspirational knowledge of so many topics within the field of informatics, and for providing me invaluable intuition during my research. I always felt confident of being in good hands throughout my supervision, and am thankful for the time and effort James has put into making me the researcher I am today, as well as for the valuable feedback provided for this thesis.

Next I would like to thank both of my secondory supervisors. It was through Christoph Dubach, who was initially my personal tutor and then my first MSc project supervisor, that I decided to pursue a PhD at the University of Edinburgh in the first place. On Christoph's departure from the university, the role of secondary supervision was passed on to Simon Fowler. One rarely meets a person as positive and optimistic as Simon, and encounters with him always left me feeling joyous and motivated. In addition, Simon has provided useful feedback that has helped improve my work substantially. Simon helped co-author my second paper and provided valuable input by implementing the case study on relational lenses.

In addition to my supervisors, I would like to show gratitude towards my coauthor Roly Perera, who provided help developing ideas in my head into an actual paper.

Furthermore, to all of my fellow PhD colleagues, especially Stefan Fehrenbach, Frank Emrich, Catriona Wedderburn, Thomas Wright, Wen Kokke, Craig McLaughlin, Jakub Zalewski, Nicole Meng, Markus Schneider and Daniel Hillestrom: My time at the University of Edinburgh was made much more enjoyable by all the great conversations and support from all of you.

The Centre for Doctoral Training in Pervasive Parallelism provided great opportunities for meeting other people, and was an important learning resource for related research. I appreciate the effort made by the organizers of the CDT, especially Murray Cole.

I would also like to thank Jeremy Gibbons and Amir Shaikhha for agreeing to be my examiners as well as taking the time to read my thesis and providing valuable
feedback. In addition, I am grateful for all comments on my published papers provided by anonymous reviewers.

Most importantly I would like to show appreciation to my family for supporting me throughout my lengthy studies, as well as for proof reading this thesis. I am especially thankful to Casey Beall, who has been by my side, supporting me and keeping me motivated throughout my PhD.

Finally, I would like to acknowledge the European Research Council for the funding provided by the ERC Consolidator Grant Skye (grant number 682315), as well as the EPSRC Centre for Doctoral Training in Pervasive Parallelism (EP/L01503X/1).

## 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.

Chapter 3 is an extended version of the following publication:

1. R. Horn, R. Perera, and J. Cheney. Incremental relational lenses. Proceedings of the ACM on Programming Languages, 2(ICFP):1-30, 2018.

Chapters 4 and 5 extend on the following publication:

1. R. Horn, S. Fowler, and J. Cheney. Language-integrated updatable views. In 31st Symposium on Implementation and Application of Functional Languages, 2020.

Appendix E was also part of this publication, but was authored by Simon Fowler. The remaining chapters contain some content from the mentioned publications. I declare that I was the primary author of both publications.

## Table of Contents

1 Introduction ..... 1
1.1 Relational lenses by example ..... 4
1.2 Research Challenges ..... 10
1.3 Outline and contributions ..... 14
2 Background ..... 16
2.1 Database Preliminaries ..... 16
2.1.1 Relation Types and Database Schemas ..... 16
2.1.2 Relations ..... 17
2.1.3 Functional Dependencies ..... 23
2.2 Lenses ..... 26
2.3 Relational Lenses ..... 28
2.3.1 Relational Revision ..... 29
2.3.2 Table Sorts ..... 30
2.3.3 Relational Lens Primitives ..... 31
2.4 Qualified Types ..... 38
3 Incremental Relational Lenses ..... 40
3.1 Incremental framework ..... 47
3.1.1 Change Structures ..... 47
3.1.2 Delta Relations ..... 47
3.1.3 Delta-Relational Operations ..... 52
3.1.4 Optimisation Rules for Delta Operations ..... 54
3.2 Incrementalising Relational Lenses ..... 56
3.2.1 Incremental Lenses ..... 56
3.2.2 Select Lens ..... 58
3.2.3 Project Lens ..... 59
3.2.4 Join Lens ..... 60
3.2.5 Rename Lens ..... 67
3.3 Evaluation ..... 68
3.3.1 Microbenchmarks ..... 68
3.3.2 DBLP Example ..... 76
3.4 Summary ..... 80
4 Turning Abstract Sets into Concrete Predicates ..... 82
4.1 Basic Predicates ..... 84
4.2 Predicate Checks ..... 87
4.2.1 Set Equivalence ..... 87
4.2.2 Lossless Join Decomposition ..... 89
4.2.3 Default Value Check ..... 92
4.2.4 Predicate Ignores Columns ..... 95
4.2.5 Dropping Column References ..... 97
4.3 Normalisation ..... 101
$4.4 \quad$ Static, Dynamic and Hybrid Predicates ..... 105
4.4.1 Hybrid Predicates ..... 106
4.4.2 Language extension ..... 109
4.4.3 Checking Hybrid Predicates ..... 115
4.5 Summary ..... 118
5 Language Integrated Relational Lenses ..... 120
5.1 Sequential Lenses ..... 122
5.2 Lens Types ..... 125
5.2.1 Lens Primitive ..... 127
5.2.2 Select Lens ..... 128
5.2.3 Join Lens ..... 128
5.2.4 Drop Lens ..... 131
5.2.5 Rename Lens ..... 132
5.2.6 Lens Functions ..... 133
5.3 Evaluation Rules ..... 133
5.4 Translation ..... 135
5.5 Language is well-behaved ..... 144
5.6 Integration in Links ..... 148
5.7 Summary ..... 150
6 Relational Lenses as a Library ..... 151
6.1 Type-Level Programming ..... 153
6.1.1 Type level values ..... 154
6.1.2 Type Families ..... 155
6.1 .3 Type Classes ..... 155
6.1.4 Constraints ..... 156
6.1.5 Record Types ..... 157
6.2 Relational Lens Kinds ..... 162
6.2.1 Tables ..... 162
6.2.2 Functional Dependencies ..... 163
6.2.3 Predicates ..... 165
6.2.4 Lens Sort ..... 173
6.3 Using Relational Lenses ..... 173
6.3.1 Syntax ..... 173
6.3.2 Database Connection ..... 174
6 6.3.3 Lens Constructors ..... 176
6.3.4 Lens Operations ..... 182
6.4 Summary ..... 182
7 Related Work ..... 185
7.1 Language-integrated query ..... 185
7.2 Updatable views and lenses ..... 186
7.3 Incremental computation ..... 188
7.4 Incremental lenses ..... 188
7.5 Row Type Inference ..... 189
7.6 Dependently Typed Programming ..... 191
8 Future Work ..... 193
8.1 Table Keys ..... 193
8.2 Incremental Performance ..... 194
8.3 Concurrent Database Access ..... 195
8.4 Additional Lenses ..... 196
8.5 Well behaved Lenses ..... 196
9 Conclusion ..... 198
Bibliography ..... 201
A Proofs for Chapter 2 ..... 212
A. 1 Comparison to original Relational Lenses ..... 212
A. 2 BPV to Structural Sequential Lenses ..... 215
A. 3 Equivalence of project lens definitions ..... 216
B Proofs for Chapter 3 ..... 218
B. 1 Proofs for Section 3.1 ..... 218
B. 2 Proofs for Section 3.2 ..... 235
C Proofs for Chapter 4 ..... 246
C. 1 Normalisation Proofs 4.3 ..... 246
C. 2 Hybrid Predicates (Section 4.4.1D ..... 248
D Proofs for Chapter 5 ..... 252
D. 1 Language is well-behaved ..... 252
D. 2 Naive Lens Semantics ..... 263
D. 3 Empty Drop Lens ..... 272
E Case Study: Curated Scientific Databases ..... 275
E. 1 Disease Curation Interface ..... 275
E. 2 Links Reimplementation ..... 277
E.2.1 Architecture ..... 277
E.2.2 Tables and Lenses. ..... 278
E.2.3 Model ..... 279
E. 3 Discussion ..... 281

## Chapter 1

## Introduction

Relational databases are considered the de facto standard for storing data persistently, offering a ready-to-use and high-performance method for storing and retrieving data efficiently in a broad range of contexts.

A typical web application is based on a three-tier architecture with user interaction on the client (web browser), application logic on the server, and data stored in a (typically relational) database as shown in Figure 1.1. Relational databases employ SQL query and update expressions, including projections, selections and joins, which correspond closely to familiar list comprehension operations in functional languages.

Yet programming real life web applications can be a very dry, repetitive and error-prone procedure. Most tasks the application must perform are quite similar in nature. One common source for errors is the typical requirement for such applications to interface with relational databases. Programs interacting with databases need to juggle various datatypes, while somehow communicating with the database using the query language it supports.

The impedance mismatch between database queries and conventional program-


Figure 1.1: A typical web application setup.
ming makes even simple programming tasks challenging [27]. Languages such as C\# [68], F\# [82, 20], Links [26], Java [90], and Ur/Web [24], and libraries such as Database-Supported Haskell [84], have partly overcome this challenge using language-integrated query, in which query expressions are integrated into the host language and type system.

Typed systems allow the compiler to statically check for any issues that may cause the program to have unintended behaviour during runtime, ensuring that welltyped programs cannot "go wrong" 69. In addition to preventing runtime errors, type systems can also assist the programmer during development by providing the programmer with interactive suggestions in the form of auto-completion [72]. This helps reduces the cost of application development, and helps the programmer to navigate larger code-bases. The compiler can also be used to ensure that static guarantees are provided, such as ensuring that only a fixed number of queries are generated for a database expression [22].

Language support for database programming is still incomplete. For example, views are a fundamental concept in databases [74] that are typically not supported in language-integrated query. A view is a relation defined over the database tables using a relational query. Views have many applications:

1. A materialised view can precompute query answers, avoiding expensive recomputation;
2. a security view shows just the information a user needs, while hiding confidential data [37];
3. views can be used to define the external schema of a database, presenting the data in a form convenient for a particular application, or for integrating data from different databases.

Most databases allow specifying views using a variant of table creation syntax, and views can then be queried in much the same way as regular database tables. It is therefore natural to wish to update a view; for example, making a security view updatable would make sense if the user is intended to have write access to the view data but should not have write access to the underlying table. Unfortunately, it is nontrivial to update views. Some view updates may correspond to multiple possible updates to the source tables, while others such as views computed with aggregates may not be translatable at all [29, 8]. Consider a record insertion
to a view that is computed by joining two tables. Adding a record to the view requires inserting records to both the left and right tables. If one of those tables already contains a record with the same primary key values, it would violate the primary key constraint. Most relational database systems only allow selection and projection operations in updatable views, so updating views defined using joins is not allowed.

Lenses were introduced by Foster et al. [36] as a generalisation of updatable views to arbitrary data structures. A lens for a source $S$ and view $V$ is a pair of functions get : $S \rightarrow V$ and put: $S \times V \rightarrow S$, allowing a source data structure to be projected onto the view, and the source to be updated with a new view. To ensure a lens has predictable results, a lens should be well-behaved which requires that it satisfies round-tripping laws. The first rule states that if a lens is updated with a view using put, then calling get on that lens should return the same view. The second rule states that calling put on an unchanged view should not change the database. The round-tripping laws required by well-behavedness are as follows:

$$
\operatorname{get}(\operatorname{put}(s, v))=v \quad \operatorname{put}(s, \operatorname{get}(s))=s
$$

Lenses can also be very well-behaved if they additionally satisfy the put-put rule. The put-put rule requires multiple subsequent put operations on a source to yield the same result as performing a single put operation with the last view on the same source. This rule is sometimes considered too strict, and hence not always desirable [36, 80]. Formally, the put-put rule is defined as follows:

$$
\operatorname{put}\left(p u t\left(s, v_{1}\right), v_{2}\right)=\operatorname{put}\left(s, v_{2}\right)
$$

Lenses are particularly well-suited to programming tasks where it is necessary to maintain consistency between 'the same' data stored in different places, as often arises in web programming. A great deal of research on bidirectional programming has considered this problem, especially in the functional programming community [81, 12, 48, 36, 38, 31, 49, [50, 89].

Perhaps surprisingly, relatively little of this work has considered view updates in databases. An exception is the work of Bohannon et al. [12], who proposed lens combinators for projections, selections and joins on relational data, and proved their well-behavedness. These relational lenses are defined using put functions which map the source database and an updated view to the updated source
database. Bohannon et al. s work showed that it is possible in principle for databases to support updatable views including joins, provided the type system tracks integrity constraints on the data, such as functional dependencies.

As defined by Bohannon et al. [12], relational lenses lacked any implementation. This thesis is a theoretical and practical investigation into how relational lenses can be embedded into programming languages. We introduce two implementations of relational lenses. The first implementation extends Links [26], a web programming language designed to simplify web development. Links allows the programmer to write server side code, client side code, markup language and database queries in a single language, which can then either be executed on the server or translated into either JavaScript, HTML or SQL as required. The second implements relational lenses as a Haskell library, without requiring any changes to the compiler.

### 1.1 Relational lenses by example

In this section we illustrate the use of relational lenses as integrated into Haskell. We use an example from Bohannon et al. [12] involving a small database of albums and tracks, and an updatable view that can be defined over it using relational lenses. We suppress the technical details of relational lenses and incrementalisation until later chapters.

Figure 1.2 shows the tables used by the example. The albums table has two columns album and quantity. The album column is a key for the table, meaning that there can only be a single entry for any album name and that each album name uniquely identifies the quantity available. The tracks table has the columns track, date, rating and album, where track, album is a key for the table and album refers to the column in the albums. Each track also uniquely defines the date and rating columns.

The first task is to define primitive lenses which reference existing tables in the relational database. This is actually a concept that does not exist in the formulation of relational lenses by Bohannon et al. [12], because the lenses are transformations between entire database schemas. We start by defining the two type aliases $\tau_{\text {album }}$ and $\tau_{\text {track }}$ for record types representing an entry in each of the two tables. The record type is a named tuple type, where each column in the table becomes
albums

| album | quantity | track | date | rating | album |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 'Disintegration' | 6 |  | 'Lullaby' | 1989 | 3 | 'Galore'

Figure 1.2: The music database used by the example.
a field in the record type with its corresponding column type.

$$
\begin{aligned}
\tau_{\text {album }} & =(\text { album }: \text { string, quantity }: \text { int }) \\
\tau_{\text {track }} & =(\text { track }: \text { string, date }: \text { int, } \text { rating }: \text { int })
\end{aligned}
$$

Relational lens definitions require the specification of functional dependencies for base tables, which are constraints that indicate which attributes determine the value of other attributes. In this example, the functional dependency album $\rightarrow$ quantity for the albums table says that if two rows have the same album attribute, they must also have the same quantity value. The primitive lens constructor lens $S$ of $\tau$ with $F$, takes the relation name $S$ of the table in the database, the record type $\tau$ and the set of functional dependency constraints $F$ applying to the table.

The tracks table has a functional dependency track $\rightarrow$ date rating, which says that date and rating depend on track. This implies that any track may appear in different albums, but should have the same date and rating. (These functional dependencies are as specified by Bohannon et al. [12], but result in a database with some redundancy; however, we keep the example as is for ease of comparison with prior work.)

$$
\begin{aligned}
& \text { let lalbums }=\text { lens albums } \text { of } \tau_{\text {album }} \text { with album } \rightarrow \text { quantity } \\
& \text { let } \text { ltracks }=\text { lens tracks } \text { of } \tau_{\text {track }} \text { with } \text { track } \rightarrow \text { date,rating }
\end{aligned}
$$

We now look at how the same primitive lenses would be constructed using the Haskell library. The Haskell library uses type-level programming extensively. The code snippet below first defines the row type Albums, specifying that there should
be a column called album of type String as well as a column quantity of type Int. The row type is constructed using a type level list of tuples. With extensions to the Glasgow Haskell Compiler [92, any constructor can be promoted to a type-level constructor by prepending them with the symbol '.

```
type Albums = '[ '("album", String), '("quantity", Int)]
lalbums = prim @"albums" @Albums @'[ '["album"] --> '["quantity"]]
type Tracks = '[
    '("track", String),
    '("date", Int),
    '("rating", Int),
    '("album", String)]
ltracks = prim @"tracks" @Tracks @'[ '["track"] --> '["date", "rating"]]
```

The snippet uses the prim constructor to create both of the primitive lenses. In each case prim requires the name of the table in the database, the row type and the corresponding functional dependencies. The primitive lens for the albums table is defined using the table name albums, the row type Albums and the corresponding functional dependencies. All of these arguments are type-level arguments, which are applied using the @ symbol. A functional dependency can be constructed using the type level - -> infix operator applied to two lists of columns. In the Haskell library, lenses are defined independently of the underlying database connection. The lens for the tracks table is constructed in a similar manner.

A common workflow within a web application is to extract some view of the data from the database, associate it with a form, and map form responses to updated versions of the associated data. If the mapping from the database to the form data is defined by a lens, then the view can be fetched from the lens using the get operation. When the user submits the form, the put operation of the lens should allow us to propagate the changes to the underlying database. The snippet below adds a row to the albums table. In this snippet the curly braces construct a singleton set, while parentheses construct a record.

$$
\text { let } r s=\text { get lalbums in }
$$

$$
\begin{aligned}
& \text { let } r s=r s \cup\{(\text { album }=\text { 'Disintegration', quantity }=1)\} \text { in } \\
& \text { put lalbums } \text { with } r s
\end{aligned}
$$

The corresponding code in Haskell performs the same operations in an IO monad. This is required because the application must query the server, which is a sideeffecting operation with network communication. Consequently, both the get and put functions require a database connection value db_connect. We assume that the db_connect value is defined externally. The recs function constructs typed records from each unnamed tuple in the list. In the example, the record type is inferred but must otherwise be provided explicitly.

```
do conn <- db_connect
    -- Fetch the view
    rs <- get conn lalbums
    -- Add the new record
    let rs = rs `union` recs [("Disintegration", 1)]
    -- Update the database
    put conn lalbums rs
```

Assuming that the albums table does not already contain an album for Disintegration, it is equivalent to performing the SQL-style insert:

INSERT INTO albums (album, quantity) VALUES ('Disintegration', 1)

In practice, views are more useful for selecting subsets of data or combining tables. For example, the web application might show a form allowing updates to a single album at a time, such as 'Galore'. A select lens $\operatorname{select}_{P}$ from $e$ is a lens that filters out all records not satisfying the predicate $P$ for the underlying lens $e$. The predicate is an expression returning a boolean value, similar to a where clause in SQL. In our example we require that album = 'Galore'.
select $_{\text {album='Galore' }}{ }^{\prime}$ from lalbums

We adjust the above Haskell snippet to use the above select lens. The predicate is constructed using \#album \#= "Galore", where \#album is a label referring to the album column and \#= is the predicate equality operator. We then call get on the select lens, make any desired changes and then update the database using put on the select lens.

```
do conn <- db_connect
    -- Create a select lens which only returns tracks from the 'Galore' album
    let lens = select (#album #= "Galore") lalbums
    -- Fetch the view
    rs <- get conn lens
    -- Make some changes
    let rs = -- update rs
    -- Update the database
    put conn lens rs
```

Suppose tracks contains the entries specified in Figure 1.3 on the left. Calling get on the lens produces the view on the right, containing only the records having album $=$ 'Galore'. If the user changes the rating of the track 'Lullaby' to 4, then submits the form, an updated view is generated as shown on the right in Figure 1.4. The application can then call put on the view, which will cause the underlying tracks table to be updated with the changes to the 'Lullaby' tracks. Notice that we must change both tracks because of the functional dependency track $\rightarrow$ date rating. This will produce the updated table as shown on the left in Figure 1.4 .

Another commonly used lens is the join lens, which combines two tables. In the example database, we may wish to compute the view of all tracks including the corresponding album quantity. The natural join between the albums and tracks table returns all the records in the tracks table where a corresponding record with the same album value in the albums table exists. For each such record, the quantity value from the albums table is included in the record. The natural join of the two tables would thus be:

| track | date | rating | album | quantity |
| :---: | :---: | :---: | :---: | :---: |
| 'Lullaby' | 1989 | 3 | 'Galore' | 1 |
| 'Lullaby' | 1989 | 3 | 'Show' | 3 |
| 'Lovesong' | 1989 | 5 | 'Galore' | 1 |
| 'Lovesong' | 1989 | 5 | 'Paris' | 5 |
| 'Trust' | 1992 | 4 | 'Wish' | 4 |

The join lens does not always have unique solutions when updating the database. When a record should be removed from the output, there is sometimes a choice

| track | date | rating | album |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 'Lullaby’ | 1989 | 3 | 'Galore' | track | date | rating | album |
| 'Lullaby' | 1989 | 3 | 'Show' ${ }_{\text {c }}$ get | 'Lullaby' | $\begin{aligned} & 1989 \\ & 1989 \end{aligned}$ | 35 | 'Galore <br> ‘Galore |
| 'Lovesong' | 1989 | 5 | 'Galore' |  |  |  |  |
| 'Lovesong' 'Trust' | 1989 | 4 | 'Paris' |  |  |  |  |

Figure 1.3: Select lens example: computing the view (right) from the source (left) using get

| track | date | rating | album |  | track | date | rating | album |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 'Lullaby’ | 1989 | 4 | 'Galore' | $\stackrel{\text { put }}{\stackrel{1}{4}}$ |  |  |  |  |
| 'Lullaby' | 1989 | 4 | 'Show' |  | 'Lullaby' | 1989 | 4 | 'Galore' |
| 'Lovesong' | 1989 | 5 | 'Galore' |  | 'Lovesong' | 1989 | 5 | 'Galore' |
| 'Lovesong' | 1989 |  | 'Paris' |  | Lovesong | 198 |  | Gatore |
| 'Trust' | 1992 | 4 | 'Wish' |  |  |  |  |  |

Figure 1.4: Select lens example: computing the new source (left) using put on the new view (right) and old source (from Fig. 1.3, left). The change to the view results in two changes to the source.
between deleting it from the left or the right table. This ambiguity is solved by having join lenses with different variants. The variant that always resolves this choice by deleting from the left table is called the delete left join lens variant. We define joined as the join between the tracks and albums tables:
let joined = join_dl ltracks with lalbums

In Haskell the delete left variant is the default behaviour. Such a lens is constructed as follows:

```
joined = join ltracks lalbums
```

Finally, we introduce the drop lens. The drop lens is used to project away columns from a table. Along with the column to be removed, the lens also requires a default value that is used when inserting a new record as well as the defining columns. When updating the database, the lens tries to find a record with the same defining columns, as the value source for the dropped columns. The lens dropped is defined as joined with the date column removed. When updating the dropped lens, the lens first tries to find a record with the same track value to
determine the date value. Otherwise the default value 2021 is used.
let dropped $=$ drop date determined by (track, 2021) from joined

The Haskell library requires the programmer to specify a static record containing the default values for the dropped columns. The programmer also provides the list of defining columns and the underlying lens. The equivalent Haskell library lens is defined as follows:

```
dropped = dropl @'[ '("date", 'P.Int 2021)] @'["track"] joined
```

Type and Integrity Constraints Both views and base tables can be associated with integrity constraints, and updated views need to respect these constraints. There are three kinds of constraints:

1. The updated view should be well-typed in the usual sense. Views that have rows with extra or missing fields, or field values of the wrong types, are ruled out statically by the type system.
2. The updated view should satisfy the functional dependencies associated with the view. Thus, the functional dependency track $\rightarrow$ date rating from our example implies that we cannot change the rating or date of 'Lullaby' in one row without changing all the others to match.
3. The updated view may also need to satisfy a predicate on the rows. Views defined by lenses may have selection conditions, such as album = 'Galore'. Inserting rows with other album values, or changing this field in existing rows, is not allowed.

These constraints all originate in the definition of schemas for relational lenses introduced by Bohannon et al. [12]. The correctness properties of relational lenses rely on these integrity constraints, and if the updated view satisfies its constraints then the updated underlying table will also satisfy its own constraints.

### 1.2 Research Challenges

To make relational lenses practical, two challenges must be solved. The first relates to how well relational lenses can scale. The second is the challenge of
integrating the required type-checking rules into the target programming language, which is complicated by the rich type information required by functional dependencies and predicates.

## Scalability

The proposed definitions of get and put are state-based - they show how to compute the view from the base table state, and how to compute the entire new state of the base tables from the updated view and the old table state. These definitions suggest an obvious, if naive, implementation strategy: computing the new source table contents and replacing the old contents. This is simply impractical for any realistic database. It becomes wasteful in the common case when updates only affect relatively few records. Replacing source tables would also necessitate locking access to the affected tables for long periods, destroying any hope of concurrent access.

Luckily, replacing entire source tables with their new contents is seldom necessary. The reason is that updates to tables (and views) are often small: for example, a row might be inserted or deleted, or a single field value modified. This thesis proposes incremental relational lenses, an adaptation to the semantics of relational lenses able to address the challenge of scalability. Instead of recomputing entire tables during a put operation, only the changes to the underlying table should be computed.

We show that our incremental semantics are equivalent to the state-based lenses. As part of the work on this thesis, incremental relational lenses were implemented in both Links and Haskell. An experimental evaluation compares the incremental to the state-based semantics. The evaluation shows that the performance of incremental semantics scale better as the database size increases.

## Language Integration

Relational lenses use a refinement type to determine which lenses can be composed. The typing rules are non-trivial, and static verification is useful to prevent unexpected behaviour during runtime. By performing compile-time verification, the lens construction is guaranteed always to succeed, preventing the programmer from having to extensively test their application for runtime errors.

Such static lens checks make relational lenses an interesting application of more advanced type systems. Just as with dynamic programming languages, it would be possible to forgo any static type checking and to defer these checks to runtime. However, relational lenses contain many checks that must be performed, and it would be easy to miss an error which may cause the application to fail. By checking them statically during compile time, issues can be fixed before the program is run.

Predicates Predicates calculate a binary outcome on a per-record basis. For the select lens, predicates determine if a record should be included in the output or not. More generally, they are used as constraints on tables, specifying which records could be present in the view. Bohannon et al. [12] define relational lens predicates using abstract sets. Any record should satisfy the predicate if and only if the record is also contained in the equivalent abstract set.

This thesis introduces a concrete predicate language as well as the corresponding checks on predicates required by the relational lens typing rules. We show how the concrete predicate language relates to abstract sets, and prove that our checks sufficiently satisfy the abstract specification. Our predicate language is trivially translatable into an SQL where-clause, allowing it to be used for querying the database server.

We then show how the predicate syntax can be extended to support function abstractions and advanced data types like tuples and records. We present a translation based on existing work by Cooper [25]. The translation normalizes the more expressive predicate into the basic predicate syntax. This approach allows us to provide a more convenient language subset to the programmer.

Another obstacle is that the predicate checks inspect the structure to determine if a predicate satisfies the necessary side-conditions. This leaves the programmer with two obvious strategies, each with their own drawbacks. The first strategy requires the predicate to be completely known during compilation, which restricts the usage scenarios of relational lenses. This option makes it impossible to implement code where the predicate is constructed based on user input. Alternatively, the predicate is only constructed during runtime, preventing the predicate checks from being performed until the lens constructor is called. Deferring the checks to runtime places the burden of checking the application behaves correctly on the
programmer, as the compiler is not able to provide these guarantees.
This thesis also introduces a third option. The checks required on predicates allow some portions of the predicate to be replaced by typing information. Hybrid predicates are predicates which are known statically by default, but allow subexpressions of the predicate to be erased so that only the type information is retained in the static portion. This allows any predicate with the same static information to be interchanged. We show that if the predicate checks can be performed on the available static predicate, the underlying dynamic predicate can safely be used by the relational lens.

Typing rules As defined by Bohannon et al. [12], a relational lens is a mapping between two database schemas. Each database schema is a mapping from relation names to tables. This representation is practical in a database setting, where the goal is to provide a whole schema to the database user. When working with relational lenses from an application programming view, there are however a few disadvantages:

- The application does not provide the relational lens with the underlying database data. Instead, the user expects the lens to interact with the database and query the data efficiently as required.
- A program typically does not concern itself with an entire database schema at once. It is more likely for one view to be queried at a time. The program should also not query views that are not required. Each database schema may potentially be quite large, and may unnecessarily pollute the lens type with unneeded information.
- Lenses may combine multiple views. The expressions producing these views can be interleaved arbitrarily, which may be less intuitive than the expression tree model used by a functional programming language.

Rather than considering lenses as bidirectional mappings between schemas, this thesis proposes representing lenses as handles to a single view. In this representation, we refer to other views using lens sub-expressions instead of relation names. In the simplest case, the sub-expression is a primitive lens referring to an existing table in the database, but it could also be any other more complicated lens as long as it is compatible.

This thesis provides a collection of typing rules for these lenses, and shows that any well-typed lens in our language can be translated into a well-typed lens as presented by Bohannon et al. [12]. The changes to relational lenses presented here help to integrate relational lenses into a functional programming language such as Links or Haskell in a more idiomatic way.

Lenses as a Library This thesis is motivated by the task of bringing relational lenses to real world applications. The first implementation extended the Links compiler to support relational lenses. The flexibility offered by extending the compiler allows the typing rules to be implemented, but increases the complexity of the compiler by adding changes specific to a single language feature.

A more ideal approach to implementing relational lenses is to implement them as a library for a language. Relational lenses require a more extensible type system to perform static checking of the typing rules. This thesis demonstrates how the Haskell type system along with extensions for type-level programming such as data kinds, constraint kinds, type operators and type applications can be used to implement statically checked relational lenses. This approach also makes it easier to abstract over expressions defining relational lenses, by deferring lens checks to the call-site when components such as the functional dependencies are unknown.

In the past, database related features have been used to justify the introduction of more advanced language features. The language C\# was extended with lambda functions and quasi-quotation to support language integrated query (LINQ) in version 3.0 [14]. Relational lenses could be a good motivation for programming languages to either support qualified types and type families or even fully dependent types.

Rename Lens Throughout this thesis we also introduce the lens for renaming columns in the view. While theoretically straightforward, this lens is important and useful for practical applications. A rename lens was not provided by Bohannon et al. [12].

### 1.3 Outline and contributions

The contributions of this thesis are outlined as follows:

- Chapter 2 presents background information required for this thesis, including fundamental concepts of relational algebra, lenses in general and an overview of relational lenses from Bohannon et al. [12].
- Chapter 3 introduces incremental relational lens semantics, including an experimental evaluation of their performance.
- Chapter 4 provides a concrete predicate syntax, and shows how the predicate checks can be performed on them. Section 4.4 presents hybrid predicates.
- Chapter 5 adapts relational lenses to the setting of functional programming languages. Lenses are presented as handles to views rather than transformations between database schemas. This section also integrates the checks on predicates defined in Section 4.2, showing that the concrete predicate syntax is well-typed.
- Chapter 6 introduces the Haskell relational lenses library. The chapter shows how type-level programming features of Haskell can be used to implement the relational lens typing rules.
- Chapter 7 discusses related work relevant to language-integrated query, updatable views and incremental computation.
- Chapter 8 discusses future work related to relational lenses.
- Chapter 9 concludes this thesis.


## Chapter 2

## Background

In this chapter we recapitulate background concepts from database theory [2] and then review the definitions of relational lenses [12]. The chapter also includes an introduction to qualified types. We use different notation from that paper in some cases, and explain the differences as necessary.

### 2.1 Database Preliminaries

### 2.1.1 Relation Types and Database Schemas

Attribute names, or simply attributes, are ranged over by $A, B, C$ and attribute values by $a, b, c$. Records $m, n$ are partial functions from attributes to attribute values. For simplicity, we assume a single (unwritten) type for attribute values; of course, in our implementation we support the usual integers, strings, booleans, etc. Records are written $(A=a, B=b, \ldots)$. Identifiers $U, V$ range over sets of attributes considered as record domains; we use $X, Y, Z$ for arbitrary sets and sometimes write $X Y$ to mean $X \cup Y$. We write $m: U$ to indicate that $\operatorname{dom}(m)=$ $U$. Basic operations on records include:

- record projection $m[V]$ : record $m$ domain-restricted to $\operatorname{dom}(m) \cap V$;
- domain antirestriction $m \backslash_{V}: m$ domain-restricted to $\operatorname{dom}(m)-V$;
- record update $m \leftarrow n: U \cup V$ : given $m: U, n: V$ defines $(m \leftarrow n)(A)$ as $n(A)$ if $A \in V$ and $m(A)$ otherwise;
- record concatenation $m \otimes n$ : a special case of record update for $m: U, n: V$,
where $U$ and $V$ are disjoint.
Given attribute $A \in U$ and $B \notin U$, we write $U[A / B]$ for $(U-\{A\}) \cup\{B\}$, and similarly if $m: U$ then $m[A / B]: U[A / B]$ is the tuple resulting from renaming attribute $A$ in $M$ to $B$. Renaming is definable as $\left(m \backslash_{\{A\}}\right) \leftarrow(B=m(A))$.


### 2.1.2 Relations

Relations $M, N, O$ are (finite) sets of records with the same domain. $M$ has domain $U$, or equivalently $M$ is a relation of type $U$, written $M: U$, if $m: U$ for all $m \in M$. Relations are closed under the standard operations of relational algebra; Figure 2.2 defines the syntax of relational expressions $q$. This includes relation constants $M$, relation names $R$, and a let construct we include for convenience. The operations,$- \cup$ and $\cap$ have their usual set-theoretic interpretation, subject to the constraint that the arguments $q, q^{\prime}$ have the same type $U$, i.e. $q, q^{\prime}: U$. Figure 2.1 lists properties on sets we rely on.

The following properties of sets in this section are also used throughout this thesis.
Lemma 1. $M \subseteq N \cup O$ iff $M-O \subseteq N$.
Proof.

First show that $M-O \subseteq N$ implies $M \subseteq N \cup O$.

$$
M-O \subseteq N
$$

$$
\Rightarrow(M-O) \cup(M \cap O) \subseteq N \cup(M \cap O) \quad \cup \text { monotone }
$$

$$
\Leftrightarrow M \subseteq N \cup(M \cap O) \quad-/ \cap \text { complementary }
$$

$$
\Rightarrow M \subseteq N \cup O \quad M \cap \text { decreasing; } \cup \text { monotone }
$$

Now show that $M \subseteq N \cup O$ implies $M-O \subseteq N$.

$$
\begin{aligned}
& M \subseteq N \cup O \\
& \Rightarrow M-O \subseteq(N \cup O)-O \\
& \Leftrightarrow M-O \subseteq(N-O) \cup(O \\
& \Leftrightarrow M-O \subseteq N-O
\end{aligned}
$$

$$
\Leftrightarrow M-O \subseteq(N-O) \cup(O-O) \quad-/ \cup \text { distr. }
$$

$$
\Rightarrow M-O \subseteq N \quad \cdot-O \text { decreasing; trans. }
$$

additive inverse

- associative
- $-N$ decreasing
- $-N$ monotone

M-. antitone
$\varnothing$ unit for $\cup$
$M \cup$ increasing
$\cup$ commutative
$\cup$ monotone
$\cup$ least upper bound
$M \cap \cdot$ decreasing
$\cap$ commutative
$\cap$ monotone
$\cap$ greatest lower bound
$\cap$ in terms of -

- and $\cap$ complementary
$\varnothing$ least
$\varnothing$ annihilator for $\cap$
$\cap$ and $\cup$ induce $\subseteq$
$\cup$ distributes over $\cap$
$\cap$ distributes over $\cup$
- distributes over $\cup$
- distributes over $\cap$
$M-M=\varnothing$
$(M-N)-O=(M-O)-N$
$M-N \subseteq M$
if $M \subseteq M^{\prime}$ then $M-N \subseteq M^{\prime}-N$
if $N \subseteq N^{\prime}$ then $M-N \supseteq M-N^{\prime}$
$M \cup \varnothing=M$
$M \subseteq M \cup N$
$M \cup N=N \cup M$
if $M \subseteq M^{\prime}$ and $N \subseteq N^{\prime}$ then $M \cup N \subseteq M^{\prime} \cup N^{\prime}$
$M \cup N \subseteq O$ iff $M \subseteq O$ and $N \subseteq O$
$M \cap N \subseteq N$
$M \cap N=N \cap M$
if $M \subseteq M^{\prime}$ and $N \subseteq N^{\prime}$ then $M \cap N \subseteq M^{\prime} \cap N^{\prime}$
$M \cap N \supseteq P$ iff $M \supseteq P$ and $N \supseteq P$
$M-(M-N)=M \cap N$
$M=(M-N) \cup(M \cap N)=(M \cup N)-(N-M)$
$\varnothing \subseteq M$
$M \cap \varnothing=\varnothing$
$M \subseteq N$ iff $M \cap N=M$ iff $M \cup N=N$
$(M \cup N) \cap P=(M \cap P) \cup(N \cap P)$
$(M \cap N) \cup P=(M \cup P) \cap(N \cup P)$
$(M \cup N)-O=(M-O) \cup(N-O)$
$(M \cap N)-O=(M-O) \cap(N-O)$

Figure 2.1: Well-known facts about sets.

Lemma 2. $(M-N)-O=M-(N \cup O)$.

Proof. Given any set $X$, we show $M-(N \cup O) \subseteq X$ iff. $(M-N)-O \subseteq X$ :

$$
\begin{array}{rlr}
M-(N \cup O) & \subseteq X & \\
\Leftrightarrow M \subseteq X \cup N \cup O & \text { Lemma } 1 \\
\Leftrightarrow M-N \subseteq X \cup O & \text { Lemma } 1 \\
\Leftrightarrow(M-N)-O \subseteq X & \text { Lemma } 1
\end{array}
$$

By instantiating $X$ with $(M-N)-O$ and $M-(N \cup O)$ it follows that the two terms are equivalent.

Lemma 3. $\left(M-M^{\prime}\right)-\left(M-N^{\prime}\right)=\left(M \cap N^{\prime}\right)-M^{\prime}$
Proof.

$$
\begin{array}{lr}
\left(M-M^{\prime}\right)-\left(M-N^{\prime}\right) & \\
\quad=\left(M-\left(M-N^{\prime}\right)\right)-M^{\prime} & - \text { associative } \\
\quad=\left(M \cap N^{\prime}\right)-M^{\prime} & \cap \text { in terms of }-
\end{array}
$$

Lemma 4. $M-N=M-(M \cap N)$.
Proof. We first show that for any $X, M-N \subseteq X$

$$
\begin{array}{rr}
M-N \subseteq X & \\
\Leftrightarrow M \subseteq X \cup N & \text { Lemma } \\
\Leftrightarrow M \subseteq(X \cup N) \cap M & M \subseteq X \Leftrightarrow M \subseteq X \cap M \\
\Leftrightarrow M \subseteq(X \cap M) \cup(N \cap M) & \cap / \cup \text { distr. } \\
\Leftrightarrow M-(N \cap M) \subseteq X \cap M & \text { Lemma } 1 \\
\Leftrightarrow M-(N \cap M) \subseteq X & \cap \text { decreasing }, M \text { upper bound }
\end{array}
$$

Lemma 5. If $M \cap N=M \cap O$, then $M-N=M-O$.

Proof.

$$
\begin{aligned}
M & -N \\
& =M-(M \\
& =M-(M \\
& =M-O
\end{aligned}
$$

$$
=M-(M \cap N) \quad \text { Lemma } 4
$$

$$
=M-(M \cap O) \quad \text { assumption }
$$

Lemma 6. $(N-M) \cap M=\varnothing$.
Proof.

$$
\begin{array}{rr}
(N-M) \cap M & \\
\quad=M-(M-(N-M)) & \cap \text { in terms of }- \\
\quad=M-M=\varnothing & (N-M) \text { disjoint from } M
\end{array}
$$

Lemma 7. If $N \cap O=\varnothing$ and $M \subseteq N \cup O$ then $M-O=N \cap M$.
Proof.

$$
\begin{array}{rlrl}
M & =M \cap(N \cup O) & M \subseteq N \cup O \\
& =(M \cap N) \cup(M \cap O) & \cap / \cup \text { distr. } \\
\Rightarrow M-O & =(M \cap N) \cup(M \cap O)-O & -O \\
& =((M \cap N)-O) \cup((M \cap O)-O) & -/ \cap \text { distr. } \\
& =((M \cap N)-O) & M \cap O \subseteq O \\
& =M \cap N & M \cap N \subseteq N, N \cap O=\varnothing
\end{array}
$$

Lemma 8. If $M \subseteq M^{\prime}$ then $(M \cup N)-M^{\prime}=N-M^{\prime}$.
Proof.

$$
\begin{array}{rlrl}
(M \cup N)-M^{\prime} & =\left(M-M^{\prime}\right) \cup\left(N-M^{\prime}\right) & -/ \cup \text { distr. } \\
& =\varnothing \cup\left(N-M^{\prime}\right) & M \subseteq M^{\prime} \\
& =N-M^{\prime} & & \text { 苼 }
\end{array}
$$

| Constant Relations | M |  |  |
| :---: | :---: | :---: | :---: |
| Queries | $q, q^{\prime}::=$ | $\begin{aligned} & M\|R\| \text { let } R=q \text { in } q^{\prime} \\ & q-q^{\prime}\left\|q \cup q^{\prime}\right\| q \cap q^{\prime} \\ & \sigma_{P}(q)\left\|\pi_{U}(q)\right\| q \bowtie q^{\prime} \mid \rho_{A / B}(q) \end{aligned}$ | Relations, names and let binding Set operations <br> Relational algebra |
| Predicates | $P, Q$ ::= | 丁 $\|\neg P\| P \wedge Q \mid P \vee Q$ | Logical connectives |
|  | \| | $A=B\|A=a\| X \in q$ | Tuple predicates |
|  | 1 | $\pi_{U}(P)\|P \bowtie Q\| \rho_{A / B}(q)$ | Relational algebra |

Figure 2.2: Syntax of relational expressions and predicates

Lemma 9. If $M \cap N=\varnothing$ then $M-\left(M^{\prime} \cup N\right)=M-M^{\prime}$.

Proof.

$$
\begin{aligned}
M-\left(M^{\prime} \cup N\right) & =M-\left(N \cup M^{\prime}\right) & \cup \text { comm. } \\
& =(M-N)-M & \text { lem. } 2 \\
& =M-M^{\prime} & M \cap N=\varnothing
\end{aligned}
$$

## Relational Algebra operators

We explain the remaining relational algebra operators in this section.
Relational projection is record projection extended to relations:

$$
\pi_{U}(M) \stackrel{\text { def }}{=}\{m[U] \mid m \in M\}
$$

Given $M: U$ and $N: V$, their natural join is defined by

$$
M \bowtie N=\{m: U \cup V \mid m[U] \in M \text { and } m[V] \in N\}
$$

$P$ ranges over predicates, which we can interpret as (possibly infinite) sets of records, or equivalently as functions from records to Booleans. Predicates are required for specifying selection filters in relational selection, as well as for the specification of filter conditions in lens definitions. We write $P: U$ to indicate a predicate over records with domain $U$. The predicate $A=B$ holds for records $m$ satisfying $m(A)=m(B)$, while $A=a$ holds when $m(A)=a$, and $U \in q$ holds when $m[U]$ is in the result of query $q$. The predicates $T$ (truth), $\neg P$ (negation),
and $P \wedge Q$ (conjunction) are interpreted as usual. For convenience we include predicates $\pi_{U}(P), P \bowtie Q$, and $\rho_{A / B}(P)$ which behave analogously to the relational operations, if we view predicates as sets of records. For example, $\pi_{U}(P)$ holds for records $u$ such that $t[U]=u$ for some $t$ satisfying $P$.

Given a predicate $P$ and relation $M$, the selection $\sigma_{P}(M)$ is defined as follows:

$$
\sigma_{P}(M) \stackrel{\text { def }}{=}\{m \in M \mid P(m)\}=M \cap P
$$

We will be interested in cases where predicates are insensitive to the values of certain attributes; we write " $P$ ignores $U$ " when $P(m)$ can be determined without considering any of the values that $m$ assigns to attributes in $U$ - i.e. when for all $m$ and $n$, if $m \backslash_{U}=n \backslash_{U}$ then $P(m) \Longleftrightarrow P(n)$.

We define the relational renaming operation $\rho_{A / B}(M)$ as

$$
\rho_{A / B}(M) \stackrel{\text { def }}{=}\{m[A / B] \mid m \in M\}
$$

which makes it possible to join tables with differing column names. As mentioned above, we also write $\rho_{A / B}(P)$ for the result of renaming attribute $A$ in predicate $P$ to $B$.

We rely on the following well-known facts about relational operators.

$$
\begin{array}{ll}
\bowtie \text { monotone } & \text { if } M \subseteq M^{\prime} \text { and } N \subseteq N^{\prime} \text { then } M \bowtie N \subseteq M^{\prime} \bowtie N^{\prime} \\
\pi_{U}(\cdot) \text { monotone } & \text { if } M \subseteq M^{\prime} \text { then } \pi_{U}(M) \subseteq \pi_{U}\left(M^{\prime}\right) \\
\sigma_{P}(M) \text { decreasing } & \sigma_{P}(M) \subseteq M
\end{array}
$$

## Other basic properties of relational operators

We also require the following properties of relational operators.
Lemma 10 ( $\bowtie$ after $\pi_{U} \times \pi_{V}$ increasing). Suppose $M: U \cup V$. Then $M \subseteq$ $\pi_{U}(M) \bowtie \pi_{V}(M)$.

Proof.

$$
\begin{array}{rlrl}
m & \in M & \text { suppose }(m)(1) \\
m[U] & \in \pi_{U}(M) & \text { (1) def. } \pi_{U}(\cdot)(2) \\
m[V] & \in \pi_{V}(M) & \text { (1) def. } \pi_{V}(\cdot)(3) \\
m & \in \pi_{U}(M) \bowtie \pi_{V}(M) & & \text { (2) (3) def. } \ltimes
\end{array}
$$

Lemma 11 ( $\pi_{U}$ after $\bowtie$ decreasing). Suppose $M: U$. Then $\pi_{U}(M \bowtie N) \subseteq M$.

Proof.
$m \in \pi_{U}(M \bowtie N) \quad$ suppose $(m)$ (1)
$m^{\prime} \in M \bowtie N \wedge m^{\prime}[U]=m \quad$ (1) def. $\pi_{U}(\cdot)$; exists $\left(m^{\prime}\right)$
$m=m^{\prime}[U] \in M$
(2) def. $\bowtie$

Lemma 12 ( $\pi$ unit). Suppose $M: U$. Then $\pi_{U}(M)=M$.
Proof. $\pi_{U}(M)=\{m[U] \mid m \in M\}=\{m \mid m \in M\}=M$.

Lemma 13 ( $\pi$ distributes over $\cup$ ).

$$
\pi_{U}(M) \cup \pi_{U}(N)=\pi_{U}(M \cup N)
$$

Proof.
$\pi_{U}(M) \cup \pi_{U}(N)$

$$
\begin{array}{ll}
=\{m[U] \mid m \in M\} \cup\{n[U] \mid n \in N\} & \text { def. } \pi_{U}(\cdot) \\
=\{m[U] \mid m \in(M \cup N)\} & \\
=\pi_{U}(M \cup N) & \text { def. } \pi_{U}(\cdot)
\end{array}
$$

### 2.1.3 Functional Dependencies

Functional dependencies are constraints restricting combinations of records. A functional dependency $U \rightarrow V$ requires that any two records with the same values for $U$ should have the same values for $V$. We say $U \rightarrow V$ is a functional dependency over $U$, written $U \rightarrow V: W$, iff $U \cup V \subseteq W$. If $U \rightarrow V$ is a functional dependency over $W$ and $M: W$, then $M$ satisfies $U \rightarrow V$, written $M \vDash U \rightarrow V$, iff $m[U]=n[U]$ implies $m[V]=n[V]$ for all $m, n \in M$. We write $m, M \vDash U \rightarrow V$ as a shorthand for $\{m\} \cup M \vDash U \rightarrow V$. It is conventional in database theory to write sets of attributes such as $\{A, B, C\}$ as $A B C$, and $A \rightarrow B C$ to mean the functional dependency $\{A\} \rightarrow\{B, C\}$.

| T-Transitivity |
| :--- |
| $M \vDash U \rightarrow V$ |$\quad M \vDash V \rightarrow W$


$M \vDash U \rightarrow W$$\quad$| T-Reflexivity |
| :--- |
| $M: U \quad V \subseteq U$ |
| $M \vDash V \rightarrow V$ |

## T-Projectivity

$\frac{M: U \quad V \subseteq U \quad W \subseteq V}{M \vDash V \rightarrow W}$

T-Additivity
$\frac{M \vDash U \rightarrow V \quad M \vDash U \rightarrow W}{M \vDash U \rightarrow V W}$
T-Augmentation
T-Pseudotransitivity

| $M \vDash U \rightarrow V \quad M \vDash V W \rightarrow U^{\prime}$ |
| :--- |
| $M \vDash U W \rightarrow U^{\prime}$ |

$$
\begin{array}{cl}
M: U^{\prime} \quad V \subseteq U^{\prime} \\
M \vDash W \rightarrow U \\
M \vDash W V \rightarrow U
\end{array} \quad \begin{aligned}
& \text { T-Split } \\
& M \vDash U \rightarrow V W \\
& M \vDash U \rightarrow V
\end{aligned}
$$

Figure 2.3: Armstrong's Axioms for functional dependencies.

Typically we work with sets $F, G$ of functional dependencies over a fixed $U$ and write $F: U$ iff $V \rightarrow W: U$ for every $V \rightarrow W \in F$. The notation $M \vDash F$ means that $M \vDash V \rightarrow W$ for all $V \rightarrow W \in F$. Likewise, $F \vDash G$ means that $M \vDash F$ implies $M \vDash G$ for any $M$, and $F \equiv G$ means that $F \vDash G$ and $G \vDash F$. We write $F[A / B]$ for the result of renaming attribute $A$ to $B$ in all functional dependencies in $F$, i.e. $F[A / B] \stackrel{\text { def }}{=}\{U[A / B] \rightarrow V[A / B] \mid U \rightarrow V \in F\}$.

Given a set of functional dependencies $F$ such that $M \vDash F$, it is possible to derive further valid functional dependencies using the inference rules by Delobel and Casey [30] and Armstrong [6] shown in Figure 2.3. Some of these rules are redundant, as it is possible to derive the full set of rules from either the rule collection:

- T-Transitivity, T-Projectivity and T-Additivity, or
- T-Reflexivity, T-Pseudotransitivity and T-Augmentation.


## Functions on Functional Dependencies

Figure 2.4 defines various functions on sets of functional dependencies. Given a set of functional dependencies $F$, the sets left $(F)$ and $\operatorname{right}(F)$ compute the union of the left or right side of all functional dependencies $X \rightarrow Y \in F$. The set of all columns referred to by a set of functional dependencies $F$ is given by names $(F)$. The set outputs $(F)$ consists of all attributes that are actually constrained in $F$ by other attributes. Finally, roots $(F)$ is the set of all nodes of $T_{F}$ that have in-degree

$$
\begin{aligned}
\operatorname{left}(F) & =\bigcup\{U \mid U \rightarrow V \in F\} \\
\operatorname{right}(F) & =\bigcup\{V \mid U \rightarrow V \in F\} \\
\operatorname{names}(F) & =\operatorname{left}(F) \cup \operatorname{right}(F) \\
\operatorname{outputs}(F) & =\bigcup\{V \mid \exists U \cdot F \vDash U \rightarrow V \text { and } U \cap V=\varnothing\} \\
\operatorname{roots}(F) & =\{U \mid \exists V \cdot U \rightarrow V \in F \text { and } U \cap \operatorname{right}(F)=\varnothing\} \\
F=\{U \rightarrow V\} \cdot F^{\prime} & \Longleftrightarrow F=\{U \rightarrow V\} \uplus F^{\prime} \text { and } U \in \operatorname{roots}(F)
\end{aligned}
$$

Figure 2.4: Operations on functional dependencies
zero.

## Functional Dependencies in Tree Form

Some operations on relational data may not have correct definitions if there are cycles among functional dependencies. Bohannon et al. [12] avoid this problem by requiring that sets of functional dependencies be in a special form called tree form. We briefly restate the definition for concreteness.

Definition 1. Given functional dependencies $F$, define

$$
V_{F}=\{U \mid U \rightarrow V \in F\} \cup\{V \mid U \rightarrow V \in F\} \quad E_{F}=\{(U, V) \mid U \rightarrow V \in F\}
$$

Then we say $F$ is in tree form if the graph $T_{F}=\left(V_{F}, E_{F}\right)$ is a forest and $V_{F}$ partitions $\cup V_{F}$.

If $F$ is in tree form, then each attribute set of $F$ corresponds to a node in a tree (or forest) where the edges correspond to elements of $F$. Moreover, no distinct nodes of $T_{F}$ have common atttibutes. For example, $\{A \rightarrow B C, B \rightarrow D\}$ is not in tree form, but is equivalent to $\{A \rightarrow B, A \rightarrow C, B \rightarrow D\}$ which is in tree form. However, $\{A \rightarrow B C, C \rightarrow A D\}$ has no equivalent tree form representation.

It is straightforward to check whether a set of functional dependencies is in tree form using a standard graph reachability algorithm. Section 6.2 shows how this can be implemented at the type level.

### 2.2 Lenses

Lenses are the most typical form of bidirectional transformations. A lens [36] $\ell: S \Leftrightarrow V$ is a transformation between two sets $S$ and $V$, where $S$ is a set of source values and $V$ is a set of possible views. The lens $\ell$ is determined by two functions, $g e t_{\ell}$ and $p u t_{\ell}$, with the following signatures:

$$
\operatorname{get}_{\ell}: S \rightarrow V \quad \text { put }_{\ell}: S \times V \rightarrow S
$$

Consider the example where the source data structure is a tuple of type $W \times V$. A program may be required to do some processing on the second component of the tuple. We could perform the required task by using a lens of type $\operatorname{proj}_{2}$ : $W \times V \Leftrightarrow V$. The forward direction of this lens could be defined by the snd function, while $p u t_{\text {proj }_{2}}$ function should reconstruct the tuple while replacing the second component with the updated view.

$$
\begin{aligned}
g e t_{\operatorname{proj}_{2}} & =\lambda(w, v) \cdot v \\
p u t_{\operatorname{proj}_{2}} & =\lambda(w, v) \cdot \lambda v^{\prime} \cdot\left(w, v^{\prime}\right)
\end{aligned}
$$

A lens is well-behaved if it satisfies two round-tripping properties relating get $_{\ell}$ and put $_{\ell}$. The correctness property PutGet ensures that whatever data we put into a lens is returned unchanged if we get it again. The hippocraticness property GetPut ensures that if we put the view value back into the lens unchanged, the underlying source value is also unchanged [80].

$$
\begin{align*}
& \operatorname{get}_{\ell}\left(p u t_{\ell}(s, v)\right)=v  \tag{PutGet}\\
& \operatorname{put}_{\ell}\left(s, \operatorname{get}_{\ell}(s)\right)=s \tag{GetPut}
\end{align*}
$$

In addition to the two round-tripping laws, we also would like the two functions to be total, ensuring that for any well-typed input the function yields a well-typed output. From now on the notation $\ell: S \Leftrightarrow V$ means that $\ell$ is a well-behaved lens from $S$ to $V$.

A consequence of requiring well-behavedness is that the codomain of get $_{\ell}$ must $^{\text {mus }}$ exactly equal the domain of the updated view $Y$ for put $_{\ell}(Y, X)$. To see why this is, we consider a simple example of lens for which we define the get function as:

$$
\operatorname{get}(X)=X \bmod 10
$$

Regardless of the exact semantics of $\operatorname{put}(Y, X)$ for this example, the lens cannot be considered well-behaved if we try to call $\operatorname{put}(15, X)$. We know that regardless of how $\operatorname{put}(Y, X)$ is implemented, GetPut will be violated because:

$$
\operatorname{get}(p u t(15, X)) \neq 15
$$

A solution to this problem is to restrict the domain for $Y$ of the put function to $0 \leq Y<10$. In essence we need something like a refinement type [40] to ensure that only valid outputs can be used as inputs. For relational lenses this means that any restrictions filtering rows in the output needs to be reflected as a requirement on the domain of the view.

Lenses form the arrows of a category, whose objects are the sources and views. The identity lens $i d_{X}: X \Leftrightarrow X$ is given by the functions get and put defined as:

$$
\operatorname{get}_{i d}(x)=x \quad \text { put }_{i d}\left(x, x^{\prime}\right)=x^{\prime}
$$

We omit the subscript on $i d$ when clear from context. The identity lens is trivially well-behaved. Diagram-order composition $\ell_{1} ; \ell_{2}: X \Leftrightarrow Z$ of the lenses $\ell_{1}: X \Leftrightarrow Y$ and $\ell_{2}: Y \Leftrightarrow Z$ is given by the functions get and put defined as:

$$
\operatorname{get}_{\ell_{1} ; \ell_{2}}(x)=\operatorname{get}_{\ell_{2}}\left(\operatorname{get}_{\ell_{1}}(x)\right) \quad \text { put }_{\ell_{1} ; \ell_{2}}(x, z)=\operatorname{put}_{\ell_{1}}\left(x, \operatorname{put}_{\ell_{2}}\left(\operatorname{get}_{\ell_{1}}(x), z\right)\right)
$$

## Product Lenses

As discussed in previous work [49, 50], the category of lenses also has symmetric monoidal products; that is, there is a construction $\otimes$ on its objects such that $X \otimes Y$ is the set of pairs $\{(x, y) \mid x \in X, y \in Y\}$, and which satisfies symmetry and associativity laws:

$$
\begin{align*}
X \otimes Y & \cong Y \otimes X  \tag{Sym}\\
X \otimes(Y \otimes Z) & \cong(X \otimes Y) \otimes Z \tag{Assoc}
\end{align*}
$$

These laws are witnessed by (invertible) lenses $s y m_{X, Y}$ and $a s s o c_{X, Y, Z}$, defined as follows:

$$
\begin{array}{r}
\operatorname{get}_{\text {sym }}(x, y)=(y, x) \\
\text { put }_{\text {sym }}\left(\_,(y, x)\right)=(x, y)
\end{array}
$$

$$
\operatorname{get}_{a s s o c}(x,(y, z))=((x, y), z)
$$

$$
\text { put }_{\text {assoc }}(-,((x, y), z))=(x,(y, z))
$$

In addition, we have the following combinator for combining two lenses 'side-byside':

$$
\begin{aligned}
\operatorname{get}_{\ell_{1} \otimes \ell_{2}}\left(x_{1}, x_{2}\right) & =\left(\operatorname{get}_{\ell_{1}}\left(x_{1}\right), \operatorname{get}_{\ell_{2}}\left(x_{2}\right)\right) \\
\operatorname{put}_{\ell_{1} \otimes \ell_{2}}\left(\left(x_{1}, x_{2}\right),\left(y_{1}, y_{2}\right)\right) & =\left(\operatorname{put}_{\ell_{1}}\left(x_{1}, y_{1}\right), \text { put }_{\ell_{2}}\left(x_{2}, y_{2}\right)\right)
\end{aligned}
$$

so that if $\ell_{1}: X_{1} \Leftrightarrow Y_{1}$ and $\ell_{2}: X_{2} \Leftrightarrow Y_{2}$ then $\ell_{1} \otimes \ell_{2}: X_{1} \otimes X_{2} \Leftrightarrow Y_{1} \otimes Y_{2}$.
These lens constructs preserve well-behavedness as characterised by the following inference rules:

$$
\begin{gathered}
\frac{\ell_{X}: X \Leftrightarrow X}{} \text { T-ID } \frac{\ell_{1}: X \Leftrightarrow Y}{\ell_{1} ; \ell_{2}: X \Leftrightarrow Z} \quad \ell_{2}: Y \Leftrightarrow Z \\
\frac{\operatorname{sym}_{X, Y}: X \otimes Y \Leftrightarrow Y \otimes X}{} \text { T-SYM } \\
\frac{a_{s s o c}^{X, Y, Z}}{}: X \otimes(Y \otimes Z) \Leftrightarrow(X \otimes Y) \otimes Z \\
\text { T-ASSOC } \\
\frac{\ell_{1}: X_{1} \Leftrightarrow Y_{1} \quad \ell_{2}: X_{2} \Leftrightarrow Y_{2}}{\ell_{1} \otimes \ell_{2}: X_{1} \otimes X_{2} \Leftrightarrow Y_{1} \otimes Y_{2}} \text { T-Product }
\end{gathered}
$$

### 2.3 Relational Lenses

Relational lenses are bidirectional transformations on relational data. Bohannon et al. [12] define relational lenses as transformations between two database schemas, where each database schema is a mapping from relation names to table types. Relational lenses are constructed by composing multiple individual lens primitives, which can perform operations such as relational joins, projections and selections.

Our presentation differs from that of Bohannon et al. [12] in that we consider relational lenses to be transformations between structural schemas. A structural schema is a tensor product of relation types. We use generic lens combinators arising from the symmetric monoidal product structure to deal with linearity. This makes it possible for each primitive to mention only the affected source and target data and not the rest of the database instance. We discuss the Bohannon
et al. [12] presentation in more detail in Chapter 5. compare the two notations in Appendix A.1, and show how such lenses can be converted into our presentation in Appendix A. 2 .

In this section we first introduce the additional relational operations required by the relational lens semantics in Section 2.3.1. Section 2.3.2 introduces the refinement type for relations, also known as the lens sort. We then introduce the individual relational lens primitives as well as their typing rules in Section 2.3.3.

### 2.3.1 Relational Revision

A key relational lens concept introduced by Bohannon et al. [12] is relational revision. Given a set of functional dependencies $F: U$ and relations $M, N: U$ such that $N \models F$, relational revision modifies $M$ to $M^{\prime}$ so that $M^{\prime} \cup N \models F$ so that all information in $M$ is preserved unless overridden by a functional dependency in $N$. For example, given $F=\{A \rightarrow B\}$ and $M=\{(A=1, B=2),(A=2, B=3)\}$ and $N=\{(A=1, B=42)\}$, the result of revising $M$ to be consistent with $N$ and $F$ is $\{(A=1, B=42),(A=2, B=3)\}$.

## Revision and Merge Operations

Relational revision is expressed in terms of a record revision operation recrevise ${ }_{F}(m, N)$ which takes a set of functional dependencies $F: U$ in tree form, a record $m: U$, and a set of records $N: U$ such that $N \models F$, and is defined by recursion over the tree structure of $F$. If $F$ is empty, record revision simply returns $m$. Otherwise, there must be at least one functional dependency $X \rightarrow Y$ in $F$ such that $X$ is a root. If $m$ and some $n \in N$ have the same values for $X$, we return $m \leftarrow n[Y]$, that is, a copy of $m$ whose $Y$ attributes have been updated with those from $n[Y]$; otherwise we return $m$ unchanged. We then recursively process the remaining functional dependencies.

The tree of $F$ is not unique, but provided $F$ is in tree form, the end result of record revision is uniquely defined, because each attribute in $\operatorname{right}(F)$ is modified at most once and no attribute can be modified until all other attributes it depends on have been modified.

Definition 2 (Relational revision). Figure 2.5 defines the relational revision operation revise $F^{( }(M, N)$ that takes two sets of records $M: U$ and $N: U$ where $N \vDash F$,

$$
\begin{aligned}
\operatorname{recrevise}_{\varnothing}(m, N) & =m \\
\operatorname{recrevise}_{\{X \rightarrow Y\} \cdot F}(m, N) & = \begin{cases}\operatorname{recrevise}_{F}(m \leftarrow n[Y], N) & \text { if } \exists n \in N . m[X]=n[X] \\
\text { recrevise }_{F}(m, N) & \text { otherwise }\end{cases} \\
\operatorname{revise}_{F}(M, N) & =\left\{\operatorname{recrevise}_{F}(m, N) \mid m \in M\right\} \\
\operatorname{merge}_{F}(M, N) & =\operatorname{revise}_{F}(M, N) \cup N
\end{aligned}
$$

Figure 2.5: Relational revision and relational merge
and applies record revision to every record $m \in M$ using the given functional dependencies $F$.

Definition 3 (Relational merge). Figure 2.5 also defines the relational merge operation merge $F(M, N)$, where $N \vDash F$, which revises $M$ according to $F$ and $N$ and then unions the result with $N$.

### 2.3.2 Table Sorts

A consequence of the roundtripping guarantees of lenses is that the output domain for a view must exactly equal the input domain of the put operation. For relational lenses, this means that the sort of a lens view must describe the restrictions that the lens may introduce.

We previously considered tables $M: U$, where the table $M$ has the domain $U$. In practice we require a more precise refinement type to describe exactly which records may appear in a table. Given a table $M$ such that $M: \operatorname{Rel}(U, P, F)$, where $\operatorname{Rel}(U, P, F)$ is the refinement type, we require that:

- The record type $U$ describes the columns and column types used by records in the relation. We require $M: U$.
- The Predicate $P$ is an arbitrary row-level integrity constraint requiring that each record in the table should satisfy some restriction. Bohannon et al. [12] use abstract sets to describe the predicate, requiring $M \vDash P$. In Chapter 4 we introduce a concrete predicate syntax, and require $P(m)$ to hold for any $m \in M$.
- The Functional dependencies $F$ are table-level integrity constraints, which

Relational lenses

```
\(\ell, \ell^{\prime}::=\operatorname{select}_{P} \mid \operatorname{drop} A\) determined by \((X, a) \mid\) join_dl \(\mid\) join \(_{P_{d}, Q_{d}} \mid\) rename \(_{A / B}\)
    \(\mid \quad\) id \(\left|\ell_{1} ; \ell_{2}\right|\) sym \(\mid\) assoc \(\mid \ell_{1} \otimes \ell_{2}\)
```

Figure 2.6: Syntax of relational lens expressions
can specify that all rows with the same values in some columns should have identical values in other columns. We require $M \vDash F$.

Consider the following example where lens composition is not allowed. We would like to construct a lens that drops the column age, and inserts a default value 0 if the age is otherwise unknown. If this lens is composed on top of a lens that requires $a g e>20$, any row with $a g e=0$ would not be permitted. It is necessary for the application to reject the composition of these lenses.

Bohannon et al. [12] indirectly attach lens refinement types to relation names by using a sort relation. By specifying that sort $(S)=(U, P, F)$ we require that if the relation name $S$ maps to the relation $M$ then $M: \operatorname{Rel}(U, P, F)$.

### 2.3.3 Relational Lens Primitives

In this section we recapitulate the primitive relational lenses introduced by Bohannon et al. [12]: a selection lens that corresponds to selection, a drop lens that corresponds to projection, and a join lens that corresponds to relational join. ("Corresponds" means that the get direction coincides with the relational operation.) We also introduce a trivial rename lens corresponding to the relational renaming operator. The syntax of the relational lenses, including the generic operations from Section 2.2, is given in Figure 2.6.

Relational lens expressions are subject to a typing judgement given in Figure 2.7, well-typed lenses are guaranteed to be well-behaved. The preconditions in these rules are those given by Bohannon et al. [12], to which we refer the reader for further explanation.

## Select Lens

The lens $\ell=\operatorname{select}_{P}: \operatorname{Rel}(U, Q, F) \Leftrightarrow \operatorname{Rel}(U, P \wedge Q, F)$ is defined in Figure 2.8 by the functions get and put.
$\frac{F \text { is in tree form } \quad Q \text { ignores outputs }(F)}{\text { select }_{P}: \operatorname{Rel}(U, Q, F) \Leftrightarrow \operatorname{Rel}(U, P \cap Q, F)}$ T-Select

$$
P=\pi_{U-A}(P) \bowtie \pi_{A}(P) \quad(A=a) \in \pi_{A}(P)
$$

drop $A$ determined by $(X, a): \operatorname{Rel}(U, P, F \uplus\{X \rightarrow A\}) \Leftrightarrow \operatorname{Rel}\left(U-A, \pi_{U-A}(P), F\right)$

$$
G \vDash U \cap V \rightarrow V
$$

$F$ is in tree form $\quad G$ is in tree form
$\frac{P \text { ignores outputs }(F) \quad Q \text { ignores outputs }(G)}{\text { join_dl }: \operatorname{Rel}(U, P, F) \otimes \operatorname{Rel}(V, Q, G) \Leftrightarrow \operatorname{Rel}(U \cup V, P \bowtie Q, F \cup G)}$ T-JoinDL

$$
G \vDash U \cap V \rightarrow V \quad P_{d}: U \cup V \quad Q_{d}: U \cup V
$$

$$
F \text { is in tree form } \quad G \text { is in tree form }
$$

$$
\frac{P \text { ignores outputs }(F) \quad Q \text { ignores outputs }(G)}{\operatorname{join}_{P_{d}, Q_{d},:} \operatorname{Rel}(U, P, F) \otimes \operatorname{Rel}(V, Q, G) \Leftrightarrow \operatorname{Rel}(U \cup V, P \bowtie Q, F \cup G)} \text { T-Join }
$$

$$
\frac{A \in U \quad B \notin U}{\text { rename }_{A / B}: \operatorname{Rel}(U, P, F) \Leftrightarrow \operatorname{Rel}\left(U[A / B], \rho_{A / B}(P), F[A / B]\right)} \text { T-Rename }
$$

Figure 2.7: Typing rules for relational lens primitives

$$
\begin{aligned}
\operatorname{get}_{\ell}(M)= & \sigma_{P}(M) \\
\operatorname{put}_{\ell}(M, N)= & \text { let } M_{0}=\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), N\right) \text { in } \\
& \text { let } N_{\#}=\sigma_{P}\left(M_{0}\right)-N \text { in } \\
& M_{0}-N_{\#}
\end{aligned}
$$

Figure 2.8: The select lens definition.

The put operation first calculates $M_{0}$, the set of records $\sigma_{\neg P}(M)$ excluded from the original view, revised to be consistent with the functional dependencies witnessed by the updated view $N$ together with $N$ itself. The set $N_{\#}$ collects records matching $P$ but not in $N$, which are removed from $M_{0}$ in order to satisfy PutGet.

## Project Lens

The lens $\ell=\operatorname{drop} A$ determined by $(X, a): \operatorname{Rel}(U, P, F \uplus\{X \rightarrow A\}) \Leftrightarrow \operatorname{Rel}(U-$ $\left.A, \pi_{U-A}(P), F\right)$ is defined in Figure $2.9{ }^{1}$ by the functions get and put.

$$
\begin{aligned}
\operatorname{get}_{\ell}(M)= & \pi_{U-A}(M) \\
\operatorname{put}_{\ell}(M, N)= & \operatorname{let} M^{\prime}=N \bowtie\{(A=a)\} \text { in } \\
& \text { revise }_{X \rightarrow A}\left(M^{\prime}, M\right)
\end{aligned}
$$

Figure 2.9: The project lens definition.

For put, each row in $N$ is initially given the default value $a$ for $A . M$ is then used to override the default value in $M^{\prime}$ using relational revision, so that if there is an entry $m \in M^{\prime}$ with the same value for the determining column $X$ the corresponding $A$ value from $M$ is used instead.

## Join Lens

Bohannon et al. [12] described several variants of lenses for join operations. All three perform the natural join of their two input relations in the get direction, but differ in how deletions are handled in the put direction. A view tuple deletion could be translated to a deletion in the left, right, or both source relations, and so there are three combinators join_dl, join_dr, and join_both expressing these

[^0]three alternatives. The typing rules for join_dr and join_both are left out, as they are identical to join_dl. In their extended report, Bohannon et al. [12] showed how to define all three combinators as a special case of a generic template. We first present the semantics for the delete left lens, and then show how this can be derived from the template.

The 'join/delete left' lens $\ell=$ join_dl : $\operatorname{Rel}(U, P, F) \otimes \operatorname{Rel}(V, Q, G) \Leftrightarrow \operatorname{Rel}(U \cup$ $V, P \bowtie Q, F \cup G)$ is given by the functions get and put defined in Figure 2.10.

$$
\begin{aligned}
\operatorname{get}_{\ell}(M, N)= & M \bowtie N \\
\operatorname{put}_{\ell}((M, N), O)= & \text { let } M_{0}=\operatorname{merge}_{F}\left(M, \pi_{U}(O)\right) \text { in } \\
& \text { let } N^{\prime}=\operatorname{merge}_{G}\left(N, \pi_{V}(O)\right) \text { in } \\
& \text { let } L=\left(M_{0} \bowtie N^{\prime}\right)-O \text { in } \\
& \text { let } M^{\prime}=M_{0}-\pi_{U}(L) \text { in } \\
& \left(M^{\prime}, N^{\prime}\right)
\end{aligned}
$$

Figure 2.10: The join lens definition.

The intuition for the put direction is as follows. We first compute $M_{0}$ by merging the projection $\pi_{U}(O)$ into source table $M$, and likewise $N^{\prime}$, merging the projection $\pi_{V}(O)$ into the source table $N$. We next identify those tuples $L$ which are in the join of $M_{0}$ and $N^{\prime}$ but which are not present in the updated view $O$. To satisfy PutGet, we must make sure these tuples do not appear in the join after updating the source relations. It is sufficient to delete each of those records from one of the two source tables; since this lens deletes uniformly from the left table, we compute $M^{\prime}$ by subtracting the projection $\pi_{U}(L)$ from $M_{0}$. Finally, $M^{\prime}$ and $N^{\prime}$ are the new values for the source tables.

The 'join/template' lens join ${ }_{P_{d}, Q_{d}}: \operatorname{Rel}(U, P, F) \otimes \operatorname{Rel}(V, Q, G) \Leftrightarrow \operatorname{Rel}(U \cup V, P \bowtie$ $Q, F \cup G)$ is given by the functions get and put defined as follows:

$$
\begin{aligned}
\operatorname{get}(M, N)= & M \bowtie N \\
\operatorname{put}((M, N), O)= & \text { let } M_{0}=\operatorname{merge}_{F}\left(M, \pi_{U}(O)\right) \text { in } \\
& \text { let } N_{0}=\operatorname{merge}_{G}\left(N, \pi_{V}(O)\right) \text { in } \\
& \text { let } L=\left(M_{0} \bowtie N_{0}\right)-O \text { in } \\
& \text { let } L_{a}=L \bowtie \pi_{U \cap V}(O) \text { in } \\
& \text { let } L_{l}=L-L_{a} \text { in } \\
& \text { let } M^{\prime}=M_{0}-\pi_{U}\left(L_{l} \cup \sigma_{P_{d}}\left(L_{a}\right)\right) \text { in } \\
& \text { let } N^{\prime}=N_{0}-\pi_{V}\left(\sigma_{Q_{d}}\left(L_{a}\right)\right) \text { in } \\
& \left(M^{\prime}, N^{\prime}\right)
\end{aligned}
$$

Unlike the 'join/delete left' lens, the 'join/template' lens partitions the set $L$ of records that should be deleted into two sets; $L_{a}$ contains all records that can be deleted form either the left, the right or both tables. $L_{l}$ contains all records that must be deleted from the left table, because deleting them from the right table would remove other records in the resulting view. $M^{\prime}$ is then calculated by removing all records in the projection of $L_{l}$ as well as all records in the projection of $L_{a}$ that satisfy the $P_{d}$ predicate. The set called $N^{\prime}$ in the delete lens is called $N_{0}$ in this lens. Instead, $N^{\prime}$ is calculated by removing the projection of all records in $L_{a}$ that satisfy $Q_{d}$ from $N_{0}$.

The definition for $M^{\prime}$ is slightly different from the well-behaved definition presented by Bohannon et al. [12]. We show that our lens definition is equivalent to the original definition and therefore also well-behaved.

## Lemma 14.

$$
M^{\prime}=\left(M_{0}-\pi_{U}\left(\sigma_{L_{a}}\left(P_{d}\right)\right)\right)-\pi_{U}\left(L_{l}\right)=M_{0}-\pi_{U}\left(L_{l} \cup \sigma_{P_{d}}\left(L_{a}\right)\right)
$$

Proof.

$$
\begin{aligned}
M^{\prime} & =\left(M_{0}-\pi_{U}\left(\sigma_{L_{a}}\left(P_{d}\right)\right)\right)-\pi_{U}\left(L_{l}\right) \\
& =M_{0}-\left(\pi_{U}\left(\sigma_{L_{a}}\left(P_{d}\right)\right) \cup \pi_{U}\left(L_{l}\right)\right) \\
& =M_{0}-\pi_{U}\left(L_{l} \cup \sigma_{P_{d}}\left(L_{a}\right)\right)
\end{aligned}
$$

Note that for the template lens to be well-behaved, it is necessary for all records $m: U \cup V$, either $m \in P_{d}$ or $m \in Q_{d}$. This ensures that the record is deleted from at least one of the underlying tables, removing it from the output view.

Lemma 15. The lens join_dl is equivalent to joinT, $\perp$.

Proof. For this to hold, we must show that:

1. $M^{\prime}=M_{0}-\pi_{U}(L)$
2. $N^{\prime}=N_{0}$

For our proof we must first show that $L_{a} \cup L_{l}$ is $L$.
We start by showing that $L_{l}$ is a subset of $L$ :

$$
\begin{aligned}
L_{l} & =L \bowtie \pi_{U \cap V}(O) \\
& =\pi_{U \cup V}\left(L \bowtie \pi_{U \cap V}(O)\right) \\
& \subseteq L
\end{aligned}
$$

def. $L_{l}$
$\pi_{U \cap V}(\cdot)$ unit
$\pi$ after $\bowtie$ decreasing
We use this to show that $L_{a} \cap L=L_{l}$ :
$L_{l} \cap L=L_{l} \quad L_{l} \subseteq L ; \cap$ induce $\subseteq$
We can now show that $L_{a} \cup €_{l}=L$ :

$$
\begin{aligned}
L_{a} & \cup L_{l} \\
& =\left(L-L_{l}\right) \cup\left(L_{l}\right) \\
& =\left(L-L_{l}\right) \cup\left(L_{l} \cap L\right)
\end{aligned}
$$

$$
=\left(L-L_{l}\right) \cup\left(L_{l}\right) \quad \text { def. } L_{a}
$$

$$
=L \quad-\text { and } \cap \text { complementary }
$$

We can now show the required two equations required:

$$
\begin{array}{rlr}
M^{\prime} & =M_{0}-\pi_{U}\left(L_{l} \cup \sigma_{P_{d}}\left(L_{a}\right)\right) & \\
& =M_{0}-\pi_{U}\left(L_{l} \cup \sigma_{\top}\left(L_{a}\right)\right) & P_{d}=\top \\
& =M_{0}-\pi_{U}\left(L_{l} \cup L_{a}\right) & \text { def. } \sigma_{\top}(\cdot) \\
& =M_{0}-\pi_{U}(L) & L_{l} \cup L_{a}=L \\
N^{\prime} & =N_{0}-\pi_{V}\left(\sigma_{Q_{d}}\left(L_{a}\right)\right) & \\
& =N_{0}-\pi_{V}\left(\sigma_{\perp}\left(L_{a}\right)\right) & Q_{d}=\perp \\
& =N_{0}-\pi_{V}(\varnothing) & \text { def. } \sigma_{\perp}(\cdot) \\
& =N_{0} & \text { def. } \pi_{V}(\cdot)
\end{array}
$$

Using the join template we can derive the other join variants.

The 'join/delete right' lens join_dr: $\operatorname{Rel}(U, P, F) \otimes \operatorname{Rel}(V, Q, G) \Leftrightarrow \operatorname{Rel}(U \cup V, P \bowtie$ $Q, F \cup G)$ is derived by defining $P_{d}=\perp$ and $Q_{d}=\top$ yielding the following functions get and put for the delete right lens:

$$
\begin{aligned}
\operatorname{get}(M, N)= & M \bowtie N \\
\operatorname{put}((M, N), O)= & \text { let } M_{0}=\operatorname{merge}_{F}\left(M, \pi_{U}(O)\right) \text { in } \\
& \text { let } N_{0}=\operatorname{merge}_{G}\left(N, \pi_{V}(O)\right) \text { in } \\
& \text { let } L=\left(M_{0} \bowtie N_{0}\right)-O \text { in } \\
& \text { let } L_{a}=L \bowtie \pi_{U \cap V}(O) \text { in } \\
& \text { let } L_{l}=L-L_{a} \text { in } \\
& \text { let } M^{\prime}=M_{0}-\pi_{U}\left(L_{l}\right) \text { in } \\
& \text { let } N^{\prime}=N_{0}-\pi_{V}\left(L_{a}\right) \text { in } \\
& \left(M^{\prime}, N^{\prime}\right)
\end{aligned}
$$

The 'join/delete both' lens join_both : $\operatorname{Rel}(U, P, F) \otimes \operatorname{Rel}(V, Q, G) \Leftrightarrow \operatorname{Rel}(U \cup V, P \bowtie$ $Q, F \cup G)$ has $P_{d}=Q_{d}=\top$, and is given by the functions get and put:

$$
\begin{aligned}
\operatorname{get}(M, N)= & M \bowtie N \\
\operatorname{put}((M, N), O)= & \text { let } M_{0}=\operatorname{merge}_{F}\left(M, \pi_{U}(O)\right) \text { in } \\
& \text { let } N_{0}=\operatorname{merge}_{G}\left(N, \pi_{V}(O)\right) \text { in } \\
& \text { let } L=\left(M_{0} \bowtie N_{0}\right)-O \text { in } \\
& \text { let } L_{a}=L \bowtie \pi_{U \cap V}(O) \text { in } \\
& \text { let } L_{l}=L-L_{a} \text { in } \\
& \text { let } M^{\prime}=M_{0}-\pi_{U}(L) \text { in } \\
& \text { let } N^{\prime}=N_{0}-\pi_{V}\left(L_{a}\right) \text { in } \\
& \left(M^{\prime}, N^{\prime}\right)
\end{aligned}
$$

## Renaming Lens

Renaming is a theoretically trivial but practically important operation in relational algebra, since otherwise there is no way to join the $A$ field of one table with the $B \neq A$ field of another. We introduce a renaming lens rename ${ }_{A / B}$ : $\operatorname{Rel}(U, P, F) \Leftrightarrow \operatorname{Rel}\left(U[A / B], \rho_{A / B}(P), F[A / B]\right)$, provided $A \in U$ and $B \notin U$, with its get and put operations defined as follows:

$$
\begin{aligned}
\operatorname{get}(M) & =\rho_{A / B}(M) \\
\operatorname{put}\left(\_, N\right) & =\rho_{B / A}(N)
\end{aligned}
$$

Bohannon et al. [12] did not define such a lens. The get and put operations are inverses of each other, meaning that $\operatorname{get}(\operatorname{put}(M, N))=N$ and $\operatorname{put}(M, \operatorname{get}(M))=$ $M$. This makes the rename lens (very) well-behaved.

### 2.4 Qualified Types

Types provide us with information about the value of a variable during runtime. The knowledge about the variable allows the compiler to verify that there is no mismatch between what value is expected and what value is provided to a segment of code. Such verification is helpful, because it allows errors to be detected and fixed before the program is even run.

When programming languages are extended with polymorphism, the information about a variable reduces from knowing everything to only knowing its structure (e.g. a tuple of two unknown types). The result is that no actual operations on parametrically polymorphic variables can be performed (with the exception of supplied polymorphic functions).

A compromise between the all or nothing approach is to specify restrictions that apply to the type variable. An example would be to specify that the type variable represents a numeric type, but without knowing if it is actually an integer or a floating point number. By knowing the type variable is a number, it would still be permitted to perform addition, which is defined on all numeric types.

Jones [53] introduce qualified types, a generic framework for handling such restrictions. The idea is that if we know the exact type, it is easy to determine if the type satisfies the restriction, also referred to as predicates. When defining a polymorphic function, the predicate becomes a constraint that must be solved at the call-site. The call-site can either further defer this restriction or satisfy it itself.

A type schema $\sigma$ such as $\forall \alpha \beta .(\alpha, \beta) \rightarrow \alpha$ allows us to quantify types over a set of type variables. In this example we describe a function which takes a tuple and returns the first component, independent of the underlying component types. Qualified types extend type schemas by allowing types to require a predicate $\pi$ to be satisfied, written $\pi \Rightarrow \sigma$ such that we may only use the type schema if evidence for $\pi$ can be provided [53]. We write $\theta P$ for the substitution $\theta$ applied to the


Figure 2.11: Qualified types inference rules.
predicate $P$. Predicates satisfy an entailment relation $P H Q$ with the following properties:

$$
\begin{aligned}
Q \subseteq P & \Longrightarrow P H Q \\
P H Q \wedge Q H R & \Longrightarrow P H R \\
P H Q & \Longrightarrow \theta P H \theta Q
\end{aligned}
$$

## Monotonicity

Transitivity
Closure Property

Typing judgements with qualified types, written $P \mid \Gamma \vdash M: \sigma$, specify that an expression $M$ types to $\sigma$ under the type environment $\Gamma$ and the predicate assumptions $P$. Qualified types make use of two additional typing rules shown in Figure 2.11. The first, Q-Inst, allows the instantiation of the qualified type $\pi \Rightarrow \sigma$ if it is possible to derive the predicate $\pi$ from predicate assumptions $P$. The second allows the abstraction over qualified types, allowing a term that can be typed to $\sigma$ under the assumption $\pi$ to have the qualified type $\pi \Rightarrow \sigma$ without the assumption $\pi$.

Qualified types can be used as a general framework to implement various features such as type classes [87]. In the case of relational lenses we would like the sideconditions required by relational lenses to be satisfied by predicates, which can then used in polymorphic settings.

## Chapter 3

## Incremental Relational Lenses

This chapter augments the semantics of relational lenses to support an efficient evaluation strategy. Bohannon et al. [12] define the get and put directions of relational lenses as set-theoretic functions showing how to compute the new source given the old source and the updated view. The get function for each lens is always the corresponding relational algebra option, e.g. the join operator, a projection or a selection. The forward direction can easily be translated into an SQL query, allowing it to be efficiently executed on the database server.

The put direction on the other hand provides a relational algebra expression to reconstruct the underlying source. The most obvious approach to implementing the put behavior of a relational lens is to use these definitions to calculate the new source table 'from scratch' and replace the old one with the new one. Having relational lens semantics that produce fresh copies of the database tables is undesirable for two reasons.

The first reason is that the performance of recomputing the entire database does not scale well. Changes to views are small in the common case and only affect a few rows in the database. The cost of reconstructing a table will be at least linear compared to its size, making recomputation from scratch wasteful when only small parts of the database are affected. Databases often accumulate data over their lifetime, meaning that an update operation with linear complexity will eventually become very expensive.

The other issue is a violation of separation of concerns. In the naive implementation setting the client reconstructs the database, even though it is the database
server's responsibility to store and manage the database. The advantage of using a relational database server is that it can be considered a black box that takes care of answering queries and applying updates. If the update semantics relied on recomputing the entire database tables, it would be necessary to transfer the entire database to the program, which would then send the updated copy back to the database server. In addition, concurrent access would not be possible while the database is being replaced, and more computation would be required to update indexes and to recheck constraints on the unchanged sections of the database. While it may be possible to express the put direction in SQL and execute it on the database server, this would not be helpful as the expression would still recompute the tables rather than updating the existing ones.

There is a large amount of literature on the problem of incremental view maintenance [28, 44, 59, 60] addressing the problem of how to modify a materialised view to keep it consistent with changes to the source tables. The benefits of incremental evaluation are not confined to databases either: witness the growing literature on adaptive and incremental functional programming [3, 46, 18]. Indeed, the foundations of change-oriented bidirectional transformations have even been investigated previously by Hofmann et al. [50] on edit lenses, Diskin et al. [31, 32] on delta lenses, Wang et al. [89] and others. It is natural to ask whether incrementalisation can be used to make relational lenses practical.

This chapter applies incremental computation techniques to relational lenses. The goal of incremental relational lenses is to avoid the expensive recomputation of tables required by naive lenses on each view update. Instead of working on full tables, only the changes to views are translated down into changes on the underlying tables.

Figure 3.1 illustrates this approach. Given a lens $\ell$ (defined by composing several primitive lenses $\ell_{i}$ ), a source $S$, and the initial view value $V=\operatorname{get}_{\ell}(S)$, suppose $V$ is updated to $V^{\prime}$. We begin by calculating a view delta (i.e. a change set) $\Delta V=V^{\prime} \ominus V$. Here $\ominus$ is the operation that calculates a delta, mapping one value to another. Then, for each step $\ell_{i}$ of the definition of $\ell$, we translate the view delta $\Delta V_{i}$ of $V_{i}$ to a source delta $\Delta V_{i-1}$. We do this by defining an incremental version of the put operation, $\delta$ put, which takes $S$ and $\Delta V$ as arguments, and


Figure 3.1: Propagating changes through lenses from view to source.
which satisfies the following law:

$$
\operatorname{put}(S, \operatorname{get}(S) \oplus \Delta V)=S \oplus \delta \operatorname{put}(S, \Delta V)
$$

where $\oplus$ denotes the application of a delta to a value. Finally, once we have calculated the source delta $\Delta S=\Delta V_{0}$, we translate it to a sequence of SQL INSERT, UPDATE and DELETE commands.

As with delta lenses [31], this approach avoids recomputing and replacing entire tables. Moreover, it can often translate small view deltas to small source deltas. Working with small deltas reduces the amount of computation and data movement incurred. On the other hand, incremental relational lenses still may need to access the source tables to compute correct deltas. We show that this can be done efficiently by issuing auxiliary queries during delta propagation.

We have implemented incremental relational lenses in Links [26], a web programming language with comprehensive support for language-integrated queries [25, 22, 35]. Our experiments show that incremental evaluation offers dramatic performance benefits over the naive state-based approach, just as one would hope or even expect. Perhaps more importantly, we prove the correctness of our approach. The state-based relational lens definitions have a number of subtleties, and proving the correctness of their incremental versions is a nontrivial challenge. Since relational lenses use set-based rather than multiset-based semantics, recent work by Koch [59] and Cai et al. [18] on incrementalising multiset operations does not apply; instead, we build on classical work on incremental view

|  | track | date | rating | album |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| - | 'Lullaby' | 1989 | 3 | 'Galore' | track | date | rating | album |
| + | 'Lullaby' | 1989 | 4 | 'Galore' $\stackrel{\text { dput }}{=}$ - | 'Lullaby' | 1989 | 3 | 'Galore' |
| - | 'Lullaby' | 1989 | 3 | 'Show' + | 'Lullaby' | 1989 | 4 | 'Galore' |
| + | 'Lullaby' | 1989 | 4 | 'Show' |  |  |  |  |

Figure 3.2: Select lens example: using $\delta p u t$, we compute the source delta (left-hand side) from the view delta (right-hand side) and the original source (Fig. 1.3 , left).
updates [73, 42]. Incremental relational lenses are also related to edit lenses [50] and some other frameworks; we discuss the relationship in detail in Section 7.3 .

Incremental Update Example Recall the introduction example in Figure 1.4 There the update semantics would delete all five tuples of the old tracks table, and then reinsert the three unchanged tuples and the new versions of the two modified ones. The desired effect in Figure 1.4 could be accomplished using SQL UPDATE operations to change just the ratings of the 'Lullaby' tracks to 4. This would typically be more efficient (especially if there were many more unaffected rows).

Therefore, we adopt an incremental approach, as outlined before. Instead of working with the entire tables, we first compute a delta for the modified view. View deltas are represented by sets of rows that should be inserted and deleted from the view. We illustrate deltas as tables with rows annotated with ' + ' (for insertion) or '-' (for deletion). An example delta for the update shown in Figure 1.4 is shown on the left of Figure 3.2. This delta is then used to calculate a delta for the source table, as shown on the right side of Figure 3.2 .

Once we have computed the change set for the underlying tables from Figure 3.2, we can use the delta and other available information (such as table keys) to produce SQL update commands that perform the desired update. A table key is a set of columns that uniquely identify a row in a table. The table key can be determined from the functional dependencies by finding a set of columns for which the transitive closure includes all columns of the table. In the Links implementation, the programmer specifies the table key when defining a handle to the table. For our example, the columns track and album are a key, and are used in the where clause of the update commands where possible. In addition, deletions and

```
1. let joined \(=\) join_dl albums with tracks in
let \(d r o p p e d=\) drop date determined by (track, 2018) from joined in
let filtered \(=\) select \(_{\text {quantity }}>2\) from dropped in
get filtered
```

Figure 3.3: A view selectLens defined by composing join, drop and select operators.
insertions with the same table key are combined into a single update command. We can perform the needed updates using two SQL update operations, as follows:
UPDATE tracks SET date $=1989$ rating $=4$ WHERE track = 'Lullaby' AND album = 'Galore'; UPDATE tracks SET date $=1989$ rating $=4$ WHERE track = 'Lullaby' AND album = 'Show';

Composition Updatable views can be defined using relational lens primitives for dropping attributes (projection), combining data from several tables (joining) and filtering rows from tables (selection). We can combine these primitive relational lenses using the general definition of composition for lenses [36].

We extend our track example as shown in Figure 3.3 by first joining the two tables in line 1. This gives us a view joined containing all tracks and their corresponding albums and album quantities. We may then decide to discard the date attribute using a projection lens in line 2, yielding view dropped. (The drop combinator includes a default value giving a value to use when new data is inserted into the view.) Finally, in line 3 we use selection to define a view filtered retaining rows with quantity greater than 2. Figure 3.4 shows each of the lenses in blue, and along the left shows how the composite lens's get produces the table in the bottom left with the three tracks 'Lullaby', 'Lovesong' and 'Trust'. We show the intermediate views in the get direction for completeness, but it is not necessary to compute them explicitly; we can compose the get directions and extract a single SQL query to produce the final output. The query for the example in Figure 3.3 would be:

```
SELECT t1.track, t1.rating, t1.album, t2.quantity
    FROM tracks as t1
    JOIN albums as t2 ON t1.album = t2.album
    WHERE t2.quantity > 2;
```

Suppose a user then makes the changes shown in red at the bottom of Figure 3.4 Performing the update with composed lenses works similarly to the case for single lenses: for a composite lens $\ell_{1} ; \ell_{2}$ we first propagate the view delta backwards through $\ell_{2}$ to obtain a source delta, then treat that as a view delta for

| album | quantity |
| :---: | :---: | :---: |
| 'Disintegration' | 6 |
| 'Show' | 3 |
| 'Galore' | 1 |
| 'Paris' | 4 |
| 'Wish' | 5 |

\{album $\rightarrow$ quantity $\}$

| track | date | rating | album |  |
| :---: | :---: | :---: | :---: | :---: |
| 'Lullaby' | 1989 | 3 | 'Galore' |  |
| 'Lullaby' | 1989 | 3 | 'Show' |  |
|  | 'Lovesong' | 1989 | 5 | 'Galore' |
| 'Lovesong' | 1989 | 5 | 'Paris' |  |
|  | 'Trust' | 1992 | 4 | 'Wish' |

$\{$ track $\rightarrow$ date, rating $\}$

|  | album | quantity |
| :---: | :---: | :---: |
| - | 'Disintegration' | 6 |
| + | 'Disintegration' | 7 |


|  | track | date | rating | album |
| :---: | :---: | :---: | :---: | :---: |
| - | 'Lullaby' | 1989 | 3 | 'Galore' |
| + | 'Lullaby' | 1989 | 4 | 'Galore' |
| - | 'Lullaby' | 1989 | 3 | 'Show' |
| + | 'Lullaby' | 1989 | 4 | 'Show' |
| - | 'Lovesong' | 1989 | 5 | 'Paris' |
| + | 'Lovesong' | 1989 | 5 | 'Disintegration' |
| - | 'Trust' | 1992 | 4 | 'Wish' |

'Wish'

| track | date | rating | album | quantity |
| :---: | :---: | :---: | :---: | :---: |
| 'Lullaby' | 1989 | 3 | 'Galore' | 1 |
| 'Lullaby' | 1989 | 3 | 'Show' | 3 |
| 'Lovesong' | 1989 | 5 | 'Galore' | 1 |
| 'Lovesong' | 1989 | 5 | 'Paris' | 4 |
| 'Trust' | 1992 | 4 | 'Wish' | 5 |

get droped put $\Uparrow$

| track | rating | album | quantity |
| :---: | :---: | :---: | :---: |
| 'Lullaby' | 3 | 'Galore' | 1 |
|  | 'Lullaby' | 3 | 'Show' |
| 'Lovesong' | 5 | 'Galore' | 1 |
|  | 'Lovesong' | 5 | 'Paris' |
|  | 'Trust' | 4 | 'Wish' |
|  |  |  |  |


|  | track | rating | album | quantity |
| :---: | :---: | :---: | :---: | :---: |
| - | 'Lullaby' | 3 | 'Galore' | 1 |
| + | 'Lullaby' | 4 | 'Galore' | 1 |
| - | 'Lullaby' | 3 | 'Show' | 3 |
| + | 'Lullaby' | 4 | 'Show' | 3 |
| - | 'Lovesong' | 5 | 'Paris' | 4 |
| + | 'Lovesong' | 5 | 'Disintegration' | 7 |
| - | 'Trust' | 4 | 'Wish' | 5 |



Figure 3.4: An example of how an update propagates through selectLens. Changes to the view are shown at the bottom in red.
$\ell_{1}$. We calculate an initial delta by comparing the updated view with the original view for the last lens. This is shown at the bottom of Figure 3.4 comparing the original view with the updated table yields the change set shown at the bottom right.

All intermediate change sets are calculated using the previous change set and by querying the database. Since the (non-incremental) put function is defined in terms of the previous source and updated view, sometimes we need to know parts of the values of the old source or old view to calculate the incremental behaviour. Therefore, for some relational lens steps we need to run one or more queries against the database during change propagation. The select lens is an example: in order to ensure that the source update preserves the functional dependency track $\rightarrow$ date rating, we need to query the database to find out what other album/track rows might need to have their ratings updated. The drop lens step also illustrates the need for auxiliary querying, in this case to find out the dropped dates of rows that are being updated. Finally, the join lens splits the changes of the joined view into changes for the individual tables; this too may require querying the underlying data. This produces the deltas shown in the top right corner of Figure 3.4.

Finally, we convert the source deltas into SQL update commands to update the underlying tables. Again we use table key information to generate concise updates, as follows:

```
UPDATE albums SET quantity = 7 WHERE album = 'Disintegration';
```

UPDATE tracks SET date $=1989$ rating $=4$ WHERE track $=$ 'Lullaby' AND album = 'Galore';
UPDATE tracks SET date $=1989$ rating $=4$ WHERE track $=$ 'Lullaby' AND album = 'Show';
INSERT INTO tracks (track, date, rating, album)
VALUES ('Lovesong', 1989, 5, 'Disintegration');
DELETE FROM tracks WHERE track = 'Lovesong' AND album = 'Paris';
DELETE FROM tracks WHERE track = 'Trust' AND album = 'Wish';

Outline and contributions In the rest of this chapter, we present necessary background on incrementalisation, define and prove the correctness of incremental relational lenses, and empirically validate our implementation to establish practicality.

- Section 3.1 introduces the framework for incremental relational queries.
- Section 3.2 presents incremental relational lenses, along with proofs of correctness of optimised forms of their $\delta p u t$ operations.
- Section 3.3 contains an experimental evaluation of the implementation.
- Section 3.4 summarizes the results of this chapter.


### 3.1 Incremental framework

To describe the incremental behaviour of relational lenses, we need to represent changes to query results in a simple, compositional way. We adopt an approach similar to Griffin et al. [42], who model "delta relations" as disjoint pairs of relations specifying tuples to be added and removed from a relation of the same type.

### 3.1.1 Change Structures

There are many formalisms to describe changes on structures. We use a notation similar to Cai et al. [18] and our relations and delta relations form a change structure in their sense. A change structure $\hat{V}$ is a tuple $(V, \Delta, \oplus, \ominus)$, where $V$ is the base set and for any $v \in V, \Delta v$ is a change set. Cai et al. require the following properties for any change set.

- For any $v \in V$ and $d v \in \Delta v, v \oplus d v \in V$.
- For any $u, v \in V, u \ominus v \in \Delta v$.
- For any $u, v \in V, v \oplus(u \ominus v)$ equals $u$.

In the following section we derive a change set for relations.

### 3.1.2 Delta Relations

Definition 4 (Delta relation). $A$ delta relation over $U$ is a pair $\Delta M=\left(\Delta M^{+}, \Delta M^{-}\right)$ of disjoint relations $\Delta M^{+}: U$ and $\Delta M^{-}: U$. The empty delta relation $(\varnothing, \varnothing)$ is written $\varnothing$. We write $\Delta M: \Delta U$ to indicate that $\Delta M$ is a delta relation over $U$.

A delta specifies a modification to a relation: for example, if $M=\{2,3,4\}$ and $\Delta M=(\{3,5\},\{4,9\})$ then $\Delta M^{+}$specifies that $\{3,5\}$ are to be added to, and $\{4,9\}$ are to be removed from $M$, resulting in the set $\{2,3,5\}$. Note that the redundant insertion of 3 specified by $\Delta M^{+}$and the redundant deletion of 9 specified by $\Delta M^{-}$are both permitted. However, Griffin et al. [42] define a delta $\Delta M: \Delta U$ to be minimal for $M: U$ if it contains no redundant insertions or deletions of that
sort; for example, $(\{5\},\{4\})$ is the minimal delta relative to $M$ equivalent to $\Delta M$ above.

Definition 5 (Minimal delta). $\Delta M: \Delta U$ is minimal for $M: U$ iff $\Delta M^{+} \cap M=\varnothing$ and $\Delta M^{-} \subseteq M$.

Definition 6 (Implicit coercion to delta-relation). Any relation $M: U$ can be implicitly coerced to a delta-relation $M: \Delta U \stackrel{\text { def }}{=}(M, \varnothing)$ which is minimal for $\varnothing: U$.

Deltas of the same type can be combined by a composition operation $\oplus$. If the input deltas $\Delta M, \Delta N$ are minimal for $M: U$, then $\Delta M \oplus \Delta N$ is also minimal for $M$.

Definition 7 (Delta merge). For any $\Delta M, \Delta N: \Delta U$, define

$$
(\Delta M \oplus \Delta N): \Delta U \stackrel{\text { def }}{=}\left(\left(\Delta M^{+}-\Delta N^{-}\right) \cup\left(\Delta N^{+}-\Delta M^{-}\right),\left(\Delta M^{-}-\Delta N^{+}\right) \cup\left(\Delta N^{-}-\Delta M^{+}\right)\right)
$$

Implicit coercion of $M$ to the delta-relation ( $M, \varnothing$ ), combined with delta merge $\oplus$, gives rise to a notion of delta application $M \oplus \Delta M$. If $\Delta M$ is minimal then the resulting delta has an empty negative component and can be coerced back to a relation.

Lemma 16. If $\Delta M$ is minimal for $M$ then $M \oplus \Delta M=\left(M-\Delta M^{-}\right) \cup \Delta M^{+}=$ $\left(M \cup \Delta M^{+}\right)-\Delta M^{-}$.

Proof.
$\Delta M$ minimal for $M$
suppose $(M, \Delta M)$
$\Delta M^{-} \subseteq M$

$$
\begin{array}{rlr}
M \oplus \Delta M & =(M, \varnothing) \oplus\left(\Delta M^{+}, \Delta M^{-}\right) & \text {coerce } \\
& =\left(\left(M-\Delta M^{-}\right) \cup\left(\Delta M^{+}-\varnothing\right),\left(\varnothing-\Delta M^{+}\right) \cup\left(\Delta M^{-}-M\right)\right) & \text { def. } \oplus \\
& =\left(\left(M-\Delta M^{-}\right) \cup \Delta M^{+}, \Delta M^{-}-M\right) & \text { simpl. } \varnothing \\
& =\left(\left(M-\Delta M^{-}\right) \cup \Delta M^{+}, \varnothing\right) & \Delta M^{-} \subseteq M \\
& =\left(M-\Delta M^{-}\right) \cup \Delta M^{+} & \text {uncoerce } \\
& =\left(M \cup \Delta M^{+}\right)-\Delta M^{-} & \Delta M^{+} \cap \Delta M^{-}=\varnothing
\end{array}
$$

Definition 8 (Delta negate). For any $\Delta M: \Delta U$, define neg $(\Delta M): \Delta U \stackrel{\text { def }}{=}\left(\Delta M^{-}, \Delta M^{+}\right)$.

Definition 9 (Delta difference). For any $\Delta M, \Delta N: \Delta U$, define $(\Delta M \ominus \Delta N)$ : $\Delta U \stackrel{\text { def }}{=} \Delta M \oplus(\operatorname{neg}(\Delta N))$.

The implicit coercion to delta-relations gives rise to a notion of relational difference $(M \ominus N): \Delta U$, not to be confused with $(M-N): U$, which is the set difference and only removes elements in $N$ from $M . M \ominus N$ can be used, for example, to calculate the difference between two views, such that $N \oplus(M \ominus N)=M$.

Lemma 17. Suppose $M: U$ and $N: U$. Then $(M \ominus N): \Delta U=(M-N, N-M)$. Moreover $M \ominus N$ is minimal for $N$.

Proof.

$$
\begin{aligned}
M \ominus N & =(M, \varnothing) \ominus(N, \varnothing) & & \text { coerce } \\
& =(M, \varnothing) \oplus(\varnothing, N) & & \text { def. } \ominus \\
& =((M-N) \cup(\varnothing-\varnothing),(\varnothing-\varnothing) \cup(N-M)) & & \text { def. } \oplus \\
& =(M-N, N-M) & &
\end{aligned}
$$

Moreover:
$M-N \subseteq M$

- $-N$ decreasing
$(N-M) \cap M=\varnothing$
lem. 6 def. minimal

We write $\operatorname{Rel}(U)$ for the set of all $M$ such that $M: U$ and $\operatorname{Rel}(\Delta U)$ for the set of all $\Delta M$ such that $\Delta M: \Delta U . \Delta_{U}$ is a mapping from $\operatorname{Rel}(U)$ to the set of deltas $\Delta M$ in $\operatorname{Rel}(\Delta U)$ which are minimal for $M$. For any $\operatorname{Rel}(U)$, we define $\widehat{\operatorname{Rel}(U)}=\left(\operatorname{Rel}(U), \Delta_{U}, \oplus, \ominus\right)$. We would like to show that $\widehat{\operatorname{Rel}(U)}$ is a change structure as defined by Cai et al. [18]. We first show some required properties:

Lemma 18. Given $M: U$ and $\Delta M: \Delta U$ minimal for $M$ then $M \oplus \Delta M: U$.

Proof. Follows from Lemma 16
Lemma 19. Given $M, N: U$, then $N \ominus M \in \Delta_{U} M$.

Proof. $\Delta_{U} M$ is the set of minimal deltas for $M$. Following Lemma 17, $N \ominus M$ is minimal for $M$.

Lemma 20. Given $M, N: U$, then $M \oplus(N \ominus M)=N$.

$$
\begin{aligned}
& \text { Proof. } \\
& M \oplus(N \ominus M)
\end{aligned}
$$

$$
=M \oplus(N-M, M-N) \quad \text { Lemma } 17
$$

$$
=(M-(M-N)) \cup(N-M) \quad \text { def. } \oplus
$$

$$
=(M \cap N) \cup(N-M) \quad \cap \text { in terms of }-
$$

$$
=N \quad-\text { and } \cap \text { complementary }
$$

We can now show that $\widehat{\operatorname{Rel}(U)}$ forms a change structure:
Theorem 1. $\widehat{\operatorname{Rel}(U)}=\left(\operatorname{Rel}(U), \Delta_{U}, \oplus, \ominus\right)$ is a change structure for $\operatorname{Rel}(U)$.

Proof. Follows from Lemma 18, Lemma 19 and Lemma 20.

The following are some useful straightforward properties of deltas:
Lemma 21. Suppose $\Delta M$ minimal for $M$. Then $(M \oplus \Delta M)-M=\Delta M^{+}$.
Proof.

$$
\begin{array}{rlr}
(M \oplus \Delta M)-M & =\left(\left(M-\Delta M^{-}\right) \cup \Delta M^{+}\right)-M & \text { lem. } 16 \\
& =\left(\left(M-\Delta M^{-}\right)-M\right) \cup\left(\Delta M^{+}-M\right) & -/ \cup \text { distr. } \\
& =\Delta M^{+} & \text {simpl.; } \Delta M^{+} \cap M=\varnothing
\end{array}
$$

Lemma 22. Suppose $\Delta M$ minimal for $M$. Then $(M \cap(M \oplus \Delta M)) \ominus M=$ $\ominus \Delta M^{-}$, hence $M-(M \oplus \Delta M)=\Delta M^{-}$.

Proof.

$$
\begin{array}{rlr}
(M \cap(M \oplus \Delta M)) \ominus M & =\left(M \cap\left(\left(M-\Delta M^{-}\right) \cup \Delta M^{+}\right)\right) \ominus M & \text { lem. } 16 \\
& =\left(M \cap\left(M-\Delta M^{-}\right)\right) \ominus M & M \cap \Delta M^{+}=\varnothing \\
& =\left(M-\Delta M^{-}\right) \ominus M & \Delta M^{-} \subseteq M \\
& =\left(\left(M-\Delta M^{-}\right)-M, M-\left(M-\Delta M^{-}\right)\right) & \text {lem. } 17 \\
& =\left(\varnothing, \Delta M^{-}\right) & \text {simpl.; } \Delta M^{-} \subseteq M
\end{array}
$$

$$
\begin{aligned}
& =\ominus\left(\Delta M^{-}, \varnothing\right) \\
& =\ominus \Delta M^{-}
\end{aligned}
$$

In particular this implies that $\Delta M^{-}=M-((M \cap(M \oplus \Delta M)))=(M-M) \cup$ $(M-(M \oplus \Delta M))=M-(M \oplus \Delta M)$.

Corollary 1. If $\Delta M$ minimal for $M$ then $(M \oplus \Delta M) \ominus M=\Delta M$.

Proof. By the previous two lemmas, $(M \oplus \Delta M) \ominus M=((M \oplus \Delta M)-M, M-$ $(M \oplus \Delta M))=\left(\Delta M^{+}, \Delta M^{-}\right)=\Delta M$.

Corollary 2. If $\Delta M$ and $\Delta M^{\prime}$ are minimal for $M$ and $M \oplus \Delta M=M \oplus \Delta M^{\prime}$ then $\Delta M=\Delta M^{\prime}$.

Proof. By Corollary 1, $\Delta M=(M \oplus \Delta M) \ominus M=\left(M \oplus \Delta M^{\prime}\right) \ominus M=\Delta M^{\prime}$.

The property $(M \oplus \Delta M) \ominus M=\Delta M$ is mentioned in Cai et al. [18] but not required by their definition of change structures. It is very helpful in our setting because it implies that query expressions incrementalise in a unique, compositional way, as we show next.

Lemma 23. If $\Delta M$ is minimal for $M=\varnothing$, then $\Delta M^{-}=\varnothing$ and $M \oplus \Delta M=$ $\Delta M^{+}$.

Proof. The minimality condition requires $\Delta M^{-} \subseteq M$, requiring $\Delta M^{-} \subseteq \varnothing$. As $\varnothing$ is the lower bound for the $\subseteq$ relation, $\Delta M^{-}=\varnothing$. Using Lemma 16 we can show that

$$
M \oplus \Delta M=\left(M-\Delta M^{-}\right) \cup \Delta M^{+}=\varnothing \cup \Delta M^{+}=\Delta M^{+} .
$$

In theory it would also be possible to define the delta merge operation as a minimality enforcing function using dependent types. In this case it would be a function of type $(M: U) \rightarrow \Delta_{U}(M) \rightarrow U$, where $\Delta_{U}(M)$ ensures that the provided delta is minimal for $M$, we leave this as future work.

### 3.1.3 Delta-Relational Operations

We now consider how to incrementalise relational operations. For each relational operator, such as $\sigma_{P}(M)$ or $\operatorname{merge}_{F}(M, N)$, we would like to define an operation that translates deltas to the arguments to a delta to the result. Incremental operations with symbolic names are written with a dot, for example $\dot{\sigma}_{P}(M, \Delta M)$, while alphabetic names have their incremental counterpart written with a preceding $\delta$, for example $\delta_{\operatorname{merge}}^{F}((M, \Delta M),(N, \Delta N))$.

For each argument $M$ given to a relational operation, the incremental relational operation takes a tuple of the argument $M$ and a delta $\Delta M$ to that argument. By default the incremental operator also requires $M$, because the semantics may depend on it. Ideally, the incremental operator can be optimized to remove the dependence on $M$, because $M$ is not immediately available during computation. $M$ depends on the base tables in the database, and may be a large data set that is expensive to evaluate. In practice, a dependence on $M$ would require the application to query the database, which introduces latency to the computation. We generally differentiate operations over all arguments, but can fix any of the variables by assuming that $\Delta M$ is empty for that variable.

The notion of delta-correctness characterises when a function $\delta o p$ with a suitable signature which operates on deltas can be considered to be a valid "incrementalisation" of a non-incremental operation $o p$. As observed by Griffin et al. [42], composing incremental relational operations is easier if they are also minimalitypreserving, so we require this as the first property in our definition.

Definition 10 (Delta-correctness). For any operation op : $X_{1} \times \cdots \times X_{n} \rightarrow Y$, a delta operation $\delta$ op $:(X \times \Delta X) \times \cdots \times\left(X_{n} \times \Delta X_{n}\right) \rightarrow \Delta Y$ is delta-correct for op if for any $\Delta x_{i}$ minimal for $x_{i}$ for $1 \leq i \leq n$, we have:

1. $\delta o p\left(\left(x_{1}, \Delta x_{1}\right), \ldots,\left(x_{n}, \Delta x_{n}\right)\right)$ is minimal for $o p\left(x_{1}, \ldots, x_{n}\right)$.
2. $o p\left(x_{1} \oplus \Delta x_{1}, \ldots, x_{n} \oplus \Delta x_{n}\right)=o p\left(x_{1}, \ldots, x_{n}\right) \oplus \delta o p\left(\left(x_{1}, \Delta x_{1}\right), \ldots,\left(x_{n}, \Delta x_{n}\right)\right)$

We usually just write $\delta o p(x, \Delta x)$ for unary operations. We write $\bar{x}$ for $x_{1}, \ldots, x_{n}$, $\overline{x, \Delta x}$ for $\left(x_{1}, \Delta x_{1}\right), \ldots,\left(x_{n}, \Delta x_{n}\right)$ and $\overline{x \oplus \Delta x}$ for $x_{1} \oplus \Delta x_{1}, \ldots, x_{n} \oplus x_{n}$. We say $\overline{\Delta x}$ is minimal for $\bar{x}$ if $\Delta x_{i}$ is minimal for $x_{i}$ for $1 \leq i \leq n$. Delta-correct operations are uniquely determined by the minimality condition:

Lemma 24. If $\delta o p$ is delta-correct then $\delta o p(\overline{x, \Delta x})=o p(\overline{x \oplus \Delta x}) \ominus o p(\bar{x})$ provided
$\Delta x_{i}$ is minimal for $x_{i}$ for $i \in 1, \ldots, n$. In particular, $\delta o p(\overline{x, \varnothing})=\varnothing$.
Proof. By Lemma 17, op $\overline{x \oplus \Delta x}) \ominus o p(\bar{x})$ is minimal for $o p(\bar{x})$, and by the definition of delta-correctness and Lemma 17 we have

$$
o p(\bar{x}) \oplus \delta o p(\overline{x, \Delta x})=o p(\overline{x \oplus \Delta x})=o p(\bar{x}) \oplus(o p(\overline{x \oplus \Delta x}) \ominus o p(\bar{x}))
$$

By Corollary 2 we can conclude $\delta o p(\overline{x, \Delta x})=o p(\overline{x \oplus \Delta x}) \ominus o p(\bar{x})$.
For ease of composition, we define $o p^{\dagger}(\overline{x, \Delta x})$ as the function that returns both the updated result and the delta.

Definition 11. For any op $: \bar{X} \rightarrow Y$, define op ${ }^{\dagger}: \overline{X \times \Delta X} \rightarrow Y \times \Delta Y$ as

$$
o p^{\dagger}(\overline{x, \Delta x})=(o p(\bar{x}), \delta o p(\overline{x, \Delta x}))
$$

We say op ${ }^{\dagger}$ is delta-correct (with respect to op) when $\delta$ op is.
If $\delta o p$ is delta-correct then whenever $\overline{\Delta x}$ is minimal for $\bar{x}$, so is $\delta o p(\overline{x, \Delta x})$ for $o p(\bar{x})$. This implies composability in the following sense:

Lemma 25. If $\delta o p_{1}: \overline{X \times \Delta X} \rightarrow \Delta Y, \delta o p_{2}: Y \times \Delta Y \rightarrow Z$ are delta-correct then $\delta o p_{2} \circ \circ p_{1}^{\dagger}$ and $\mathrm{op}_{2}^{\dagger} \circ \circ p_{1}^{\dagger}$ are delta-correct (both with respect to $o p_{2} \circ o p_{1}$ ).

Proof. From our assumptions we know that for any $\overline{\Delta x}$ minimal for $\bar{x}, \delta o p_{1}(\overline{x, \Delta x})$ is minimal for $o p_{1}(\bar{x})$ and that $o p_{1}(\overline{x \oplus \Delta x})=o p_{1}(\bar{x}) \oplus \delta o p_{1}(\overline{x, \Delta x})$, as well as that for any $\Delta y$ minimal for $y$ then $\delta o p_{2}(y, \Delta y)$ is minimal for $o p_{2}(y)$ and that $o p_{2}(y \oplus \Delta y)=o p_{2}(y) \oplus \delta o p_{2}(y, \Delta y)$.

For $\delta o p_{2} \circ o p_{1}^{\dagger}$ to be correct we must show two properties:

- $\delta o p_{2}\left(o p_{1}(\bar{x}), \delta o p_{1}(x, \Delta x)\right)$ must be minimal for $o p_{2}\left(o p_{1}(\bar{x})\right)$
- op $2_{2}\left(o p_{1}(\overline{x \oplus \Delta x})\right)=o p_{2}\left(o p_{1}(\bar{x})\right) \oplus \delta o p_{2}\left(o p_{1}(\bar{x}), \delta o p_{1}(\overline{x, \Delta x})\right)$

Both of these properties follow from the fact that $\delta_{o p_{1}}(\overline{x, \Delta x})$ is minimal for $o p_{1}(\bar{x})$ and that $\delta o p_{2}$ is delta-correct. For the second property we can show that:

$$
\begin{aligned}
& o p_{\mathcal{Z}}\left(o p_{1}(\overline{x \oplus \Delta x})\right) \\
& \quad=o p_{2}\left(o p_{1}(\bar{x}) \oplus \delta o p_{1}(\overline{x, \Delta x})\right) \\
& \quad=o p_{2}\left(o p_{1}(\bar{x})\right) \oplus \delta o p_{2}\left(o p_{1}(\bar{x}), \delta o p_{1}(\overline{x, \Delta x})\right)
\end{aligned}
$$

It follows that $o p_{2}{ }^{\dagger} \circ o p_{1}^{\dagger}$ is delta-correct.

Furthermore, this implies we may incrementalise any function built up out of incrementalisable relational operations, by replacing ordinary operators with their incremental counterparts, largely as described by Cai et al. [18]. Given a query $q\left(R_{1}, \ldots, R_{n}\right)$, we can transform it to a delta-correct (but not necessarily efficient) incremental version by taking $\delta(q)=\operatorname{let}(R, \Delta R)=(q)^{\dagger}$ in $\Delta R$, where the transformation $(\cdot)^{\dagger}$ is defined as follows:

$$
\begin{aligned}
M^{\dagger} & =(M, \varnothing) \quad o p\left(q_{1}, \ldots, q_{n}\right)^{\dagger} & =\left(o p^{\dagger}\left(\left(q_{1}\right)^{\dagger}, \ldots,\left(q_{n}\right)^{\dagger}\right)\right) \\
R^{\dagger} & =(R, \Delta R) \quad \operatorname{let} R=q \operatorname{in} q^{\prime \dagger} & =\operatorname{let}(R, \Delta R)=(q)^{\dagger} \operatorname{in}\left(q^{\prime}\right)^{\dagger}
\end{aligned}
$$

Essentially $(q)^{\dagger}$ traverses the query, replacing relation variables with pairs of variables and deltas, replacing constant relations with pairs ( $M, \varnothing$ ) and dealing with individual operations and let-bindings compositionally. We abuse notation slightly by adding syntax for pairs.

The following examples shows how the translation can be used to determine an incremental version of a simple expression performing a natural join between the relation $T$ and a constant relation, followed by a projection.

$$
\begin{aligned}
& \left(\operatorname{let} S=T \bowtie\{(c=5)\} \text { in } \pi_{\{a, c\}}(S)\right)^{\dagger} \\
& =\left(\operatorname{let}(S, \Delta S)=(T, \Delta T) \bowtie^{\dagger}(\{(c=5)\}, \varnothing) \text { in } \pi_{\{a, c\}}^{\dagger}(S)\right)
\end{aligned}
$$

If we are only interested in the expression computing the delta, we project the returning expression onto the delta component. In our example $\pi_{\{a, c\}}^{\dagger}(S)=$ $\left(\pi_{\{a, c\}}(S), \dot{\pi}_{\{a, c\}}(S)\right)$, where the second component of the tuple is the delta component. The full expression computing the delta is:

$$
\text { let }(S, \Delta S)=(T, \Delta T) \bowtie^{\dagger}(\{(c=5)\}, \varnothing) \text { in } \dot{\pi}_{\{a, c\}}(S)
$$

Theorem 2. If $q: \operatorname{Rel}\left(U_{1}\right) \times \cdots \times \operatorname{Rel}\left(U_{n}\right) \rightarrow \operatorname{Rel}(U)$ then $\delta(q)$ and $(q)^{\dagger}$ are deltacorrect with respect to $q$.

Proof. Shown in Appendix B. 1

### 3.1.4 Optimisation Rules for Delta Operations

To sum up, we have established that for any query there is a an extensionally unique incrementalisation, obtained by computing the difference between
the updated query result and the original result. Of course, this is far from an efficient implementation strategy. In this section, we present a number of optimisation rules for incremental relational operations, as well as relational revision and merge.

Most of the following characterisations of incremental relational operations are presented in prior work such as Griffin et al. [42], but without detailed proofs; we include detailed proofs in the appendix.

Lemma 26. [Valid optimisations] Assume $\Delta M, \Delta N$ are minimal for $M, N$ respectively. Then:

1. $\dot{\sigma}_{P}(M, \Delta M)=\left(\sigma_{P}\left(\Delta M^{+}\right), \sigma_{P}\left(\Delta M^{-}\right)\right)$
2. $\dot{\pi}_{U}(M, \Delta M)=\left(\pi_{U}\left(\Delta M^{+}\right)-\pi_{U}(M), \pi_{U}\left(\Delta M^{-}\right)-\pi_{U}(M \oplus \Delta M)\right)$
3. $(M, \Delta M) \dot{\bowtie}(N, \Delta N)=\left(\left((M \oplus \Delta M) \bowtie \Delta N^{+}\right) \cup\left(\Delta M^{+} \bowtie(N \oplus \Delta N)\right),\left(\Delta M^{-} \bowtie\right.\right.$ $\left.N) \cup\left(M \bowtie \Delta N^{-}\right)\right)$
4. $\dot{\rho}_{A / B}(M, \Delta M)=\left(\rho_{A / B}\left(\Delta M^{+}\right), \rho_{A / B}\left(\Delta M^{-}\right)\right)$
5. If $N \subseteq M$ and $N \oplus \Delta N \subseteq M \oplus \Delta M$ then $(M, \Delta M) \dot{-}(N, \Delta N)=\Delta M \ominus \Delta N$

Proof. Shown in Appendix B.1.
Relational revision is only used directly for drop lenses, where only the first argument $M$ may change. The following lemma provides an optimisation for this case:

Lemma 27. Suppose $M \models X \rightarrow A$ and $M \oplus \Delta M \models X \rightarrow A$. Then

$$
\text { drevise }_{X \rightarrow A}((M, \Delta M),(N, \varnothing))=\left(\text { revise }_{X \rightarrow A}\left(\Delta M^{+}, N\right) \text {,revise }_{X \rightarrow A}\left(\Delta M^{-}, N\right)\right) .
$$

Proof. Shown in Appendix B.1.

For relational merge, the join lens makes use of the following special case:
Lemma 28. If $\operatorname{merge}_{F}(M, N)=M$ then

$$
\operatorname{smerge}_{F}((M, \varnothing),(N, \Delta N))=\operatorname{merge}_{F}\left(M, \Delta N^{+}\right) \ominus M .
$$

Proof. Shown in Appendix B. 1 .

In select and join lenses, we avoid explicitly recomputing $\operatorname{merge}_{F}(M, N)$ by showing that it is sufficient to consider only a subset of possibly-affected rows in $M$. We define a function called affected $_{F}$ which returns a predicate selecting a (hopefully small) superset of the rows that may be changed by relational merge according to $F$ and a set of view records $N$. The returned predicate is the necessary condition for any changes implied by $F$ and $N$. The functional dependencies are expected to be non-empty, which can always be achieved by adding a functional dependency from the entire domain to the empty set.

Definition 12. affected $_{F}(N) \stackrel{\text { def }}{=} \bigvee_{X \rightarrow Y \in F} X \in \pi_{X}(N)$.
It is then possible to replace the target relation $M$ with only those rows in $M$ which are likely to be updated, allowing fewer rows to be queried from the database:

Lemma 29. If $P=\operatorname{affected}_{F}\left(\Delta N^{+}\right)$and either $F \neq \varnothing$ or $\Delta N^{+} \cap M=\varnothing$ then

$$
\operatorname{merge}_{F}\left(M, \Delta N^{+}\right) \ominus M=\operatorname{merge}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \ominus \sigma_{P}(M) .
$$

Proof. Shown in Appendix B. 1
We use the notation $F^{* U}$ to mean either $\{U \rightarrow \varnothing\}$ if $F=\varnothing$ or $F$ otherwise.
Corollary 3. If $P=\operatorname{affected}_{F^{* U}}\left(\Delta N^{+}\right)$then

$$
\operatorname{merge}_{F}\left(M, \Delta N^{+}\right) \ominus M=\operatorname{merge}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \ominus \sigma_{P}(M) .
$$

### 3.2 Incrementalising Relational Lenses

The previous Section introduces a framework for incremental relational algebra expressions. This framework includes the notion of changes to sets as well as incremental versions of common relational algebra operations. In this Section we apply the framework to relational lenses, deriving efficient expressions that compute the delta on the input view given a delta on the output view for each lens type.

### 3.2.1 Incremental Lenses

Assume $S, V$ are sets of relations having sets of deltas $\Delta S, \Delta V$ and corresponding operations $\oplus, \ominus$. A (well-behaved) incremental lens $\ell: S \Leftrightarrow V$ is a well-behaved
lens equipped with additional operations $\delta \operatorname{get}_{\ell}: S \times \Delta S \rightarrow \Delta V$ and $\delta p u t_{\ell}: S \times$ $\Delta V \rightarrow \Delta S$ satisfying

$$
\begin{align*}
\operatorname{get}_{\ell}(s \oplus \Delta s) & =\operatorname{get}_{\ell}(s) \oplus \delta g e t_{\ell}(s, \Delta s) \\
\operatorname{put}_{\ell}\left(s, \operatorname{get}_{\ell}(s) \oplus \Delta v\right) & =s \oplus \delta p u t_{\ell}(s, \Delta v)
\end{align*}
$$

( $\Delta$ PutGet)
and such that if $\Delta s$ is minimal for $s$ then $\operatorname{\delta get}_{\ell}(s, \Delta s)$ is minimal for $\operatorname{get}_{\ell}(s)$, and likewise if $\Delta v$ is minimal for $\operatorname{get}_{\ell}(s)$ then $\delta p u t_{\ell}(s, \Delta v)$ is minimal for $s$.

The $\delta g e t_{\ell}$ direction simply performs incremental view maintenance, which is not our main concern here; we include it to show how it fits together with $\delta p u t_{\ell}$ but do not discuss it further. The first equation and minimality condition is simply delta-correctness of $\delta g e t_{\ell}$ relative to $g e t_{\ell}$.

We focus here on the $\delta p u t_{\ell}$ operation. In this direction, it would be redundant to supply an argument holding the previous value of the view, since it can be obtained via $g e t_{\ell}$. The $\Delta$ PutGet rule and associated minimality condition is a special case of the delta-correctness rule, where we only consider changes to $V$, not $S$ :

$$
\operatorname{put}_{\ell}\left(s, g e t_{\ell}(s) \oplus \Delta v\right)=\operatorname{put}_{\ell}\left(s, \operatorname{get}_{\ell}(s)\right) \oplus \delta p u t_{\ell}(s, \Delta v)
$$

and the term $\operatorname{put}_{\ell}\left(s, g e t_{\ell}(s)\right)$ has been simplified to $s$ by the GetPut rule.
We can equip the generic lens combinators from 2.2 with suitable delta-correct סput operations as follows:

$$
\begin{aligned}
& \delta p u t_{\mathrm{id}}(-, \Delta x)=\Delta x \\
& \delta p u t_{\text {sym }}(-,(\Delta y, \Delta x))=(\Delta x, \Delta y) \\
& \delta p u t_{\mathrm{assoc}(-,((\Delta x, \Delta y), \Delta z))}=(\Delta x,(\Delta y, \Delta z)) \\
& \delta p u t_{\ell_{1}, \ell_{2}}(x, \Delta z)=\delta p u t_{\ell_{1}}\left(x, \delta p u t_{\ell_{2}}\left(\text { get }_{\ell_{1}}(x), \Delta z\right)\right) \\
& \delta p u t_{\ell_{1} \otimes \ell_{2}}\left(\left(x_{1}, x_{2}\right),\left(\Delta y_{1}, \Delta y_{2}\right)\right)=\left(\delta p u t_{\ell_{1}}\left(x_{1}, \Delta y_{1}\right), \delta p u t_{\ell_{2}}\left(x_{2}, \Delta y_{2}\right)\right)
\end{aligned}
$$

It is straightforward to show that the resulting incremental lenses are well-behaved.
For each relational lens primitive $\ell$ described in 2.3 , select ${ }_{P}, \operatorname{drop} A$ determined by $(X, a)$, join_dl and rename $_{A / B}$, we will define an incremental $\delta p u t_{\ell}$ operation as follows. First, we incrementalise the corresponding $p u t_{\ell}$ definition from 2.3, obtaining a function $\delta P u t_{\ell}:(S \times \Delta S) \times(V \times \Delta V) \rightarrow \Delta S$ that is delta-correct with
respect to put $_{\ell}$. Since we are only interested in the case where $S$ does not change and $v=\operatorname{get}_{\ell}(s)$, we then specialize this operation to obtain $\delta p u t_{\ell}(s, \Delta v)=$ $\delta P u t_{\ell}\left((s, \varnothing),\left(\operatorname{get}_{\ell}(s), \Delta v\right)\right)$, which yields a well-behaved lens. We then apply further optimisations to simplify this expression to a form that can be evaluated efficiently.

The well-behavedness of the generic lens combinators and the relational lens primitives imply the well-behavedness of any well-typed lens expression.

### 3.2.2 Select Lens

The incremental lens $\ell=\delta \operatorname{select}_{P}: \operatorname{Rel}(U, Q, F) \Leftrightarrow \operatorname{Rel}(U, P \wedge Q, F)$ is the lens $\operatorname{select}_{P}$ defined in Figure 2.8 of the same type, equipped with $\delta p u t_{\ell}$. The lens is defined as follows:

$$
\begin{aligned}
& \delta \text { put }_{\ell}: \operatorname{Rel}(U, Q, F) \times \Delta \operatorname{Rel}(U, P \wedge Q, F) \rightarrow \Delta \operatorname{Rel}(U, Q, F) \\
&{\delta p u t_{\ell}(M, \Delta N)=}\left(\operatorname{let} N=\sigma_{P}(M)\right. \text { in } \\
& \operatorname{let}\left(M_{0}, \Delta M_{0}\right)=\operatorname{merge}_{F}^{\dagger}\left(\sigma_{\neg P}^{\dagger}(M, \varnothing),(N, \Delta N)\right) \text { in } \\
& \operatorname{let}\left(N_{\#}, \Delta N_{\#}\right)=\sigma_{P}^{\dagger}\left(M_{0}, \Delta M_{0}\right)-\dagger(N, \Delta N) \text { in } \\
&\left(M_{0}, \Delta M_{0}\right)-\left(N_{\#}, \Delta N_{\#}\right)
\end{aligned}
$$

We define $N$ as $\operatorname{get}_{\text {select }_{P}}(M)$. The tuple $\left(M_{0}, \Delta M_{0}\right)$ computes the changes to the records that don't match the predicate $P$. We can expect $M_{0}$ to be the same as $M$, because an unchanged view should not make changes to the underlying source. The tuple ( $N_{\#}, \Delta N_{\#}$ ) determines which records should be deleted from the source, where $N_{\#}$ can be expected to be empty. All the records that should be deleted will be contained in $\Delta N_{\#}{ }^{+}$. Finally, the put expression returns the delta difference between the updated records and the records to be removed.

Lemma 30. The incremental select lens $\delta$ select $_{P}$ is well-behaved.

Proof. Follows from Theorem 2.

Definition 13. Define an optimised incremental $\delta_{\text {select }}^{P}$ lens $\ell^{\prime}$ with $\delta$ put $_{\ell^{\prime}}$ defined as follows:

$$
\begin{aligned}
\text { dput }_{\boldsymbol{\ell}^{\prime}}(M, \Delta N)= & \text { let } Q=\operatorname{affected}_{F}\left(\Delta N^{+}\right) \text {in } \\
& \text { let } \Delta M_{0}=\left(\operatorname{merge}_{F}\left(\sigma_{Q \wedge \neg P}(M), \Delta N^{+}\right) \ominus \sigma_{Q \wedge \neg P}(M)\right) \ominus \Delta N^{-} \text {in } \\
& \text { let } \Delta N_{\#}=\left(\sigma_{P}\left(\Delta M_{0}^{+}\right), \sigma_{P}\left(\Delta M_{0}^{-}\right)\right) \ominus \Delta N \text { in } \\
& \Delta M_{0} \ominus \Delta N_{\#}
\end{aligned}
$$

The optimised version works as follows. $\Delta M_{0}$ can be calculated by querying the database for $\sigma_{Q \wedge \neg P}(M)$ and then performing relational merge using $\Delta N^{+}$. The remaining computations involve only deltas and can be performed in-memory. $\Delta M_{0}$ contains all changes to the underlying table including any removed rows, but does not account for rows which previously didn't satisfy $P$, but do after the updates. These rows, which would violate lens well-behavedness, are found in $\Delta N_{\#}$. We calculate $\Delta N_{\#}$ just using the delta difference operator $\ominus$ because $N_{\#} \oplus \Delta N_{\#}$ is always a subset of $M_{0} \oplus \Delta M_{0}$. The final update consists of the changes to the table $M_{0}$ merged with the changes to remove all rows in $\Delta N_{\#}$.

Theorem 3. [Correctness of optimised select lens] Suppose $N=\sigma_{P}(M)$ where $M: \operatorname{Rel}(U, Q, F)$. Suppose also that $\Delta N$ is minimal with respect to $N$ and that $N \oplus \Delta N: \operatorname{Rel}(U, P \wedge Q, F)$. Then $^{\delta p u t}{ }_{\ell}(M, \Delta N)=\operatorname{\delta put}_{\ell^{\prime}}(M, \Delta N)$.

Proof. Shown in Appendix B. 2

### 3.2.3 Project Lens

The incremental lens $\ell=\delta$ drop $A$ determined by $(X, a): \operatorname{Rel}(U, P, F) \Leftrightarrow \operatorname{Rel}(U-$ $\left.A, \pi_{U-A}(P), F^{\prime}\right)$, where $F \equiv F^{\prime} \uplus\{X \rightarrow A\}$, is the lens drop $A$ determined by $(X, a)$ defined in Figure 2.9 of the same type, equipped with $\delta p u t_{\ell}$ defined as follows:

$$
\begin{aligned}
& \text { dput }_{\ell}: \operatorname{Rel}(U, P, F) \times \Delta \operatorname{Rel}\left(U-A, \pi_{U-A}(P), F^{\prime}\right) \rightarrow \Delta \operatorname{Rel}(U, P, F) \\
&{\delta p u t_{\ell}(M, \Delta N)=} \operatorname{let} N=\pi_{U-A}(M) \text { in } \\
& \operatorname{let}\left(M^{\prime}, \Delta M^{\prime}\right)=(N, \Delta N) \bowtie^{\dagger}(\{\{A=a\}\}, \varnothing) \text { in } \\
& \delta^{\prime} \text { revise }_{X \rightarrow A}\left(\left(M^{\prime}, \Delta M^{\prime}\right),(M, \varnothing)\right)
\end{aligned}
$$

$N$ is defined as $p u t_{\text {drop } A \text { determined by }(X, a)}(M)$. Just as in the non-incremental version, the tuple ( $M^{\prime}, \Delta M^{\prime}$ ) computes a copy of the view extended by the column $A$ with the default value $a$. The resulting delta is computed by using record revision using the unchanged source ( $M^{\prime}, \Delta M^{\prime}$ ).

Lemma 31. The incremental projection lens ddrop A determined by $(X, a)$ is wellbehaved.

Proof. Follows from Theorem 2.

Definition 14. Define an optimised incremental $\delta d r o p ~ A ~ d e t e r m i n e d ~ b y ~(~ X, a) ~ l e n s ~$ $\ell^{\prime}$ with $\delta p u t_{\ell^{\prime}}$ defined as follows:

$$
\begin{aligned}
\delta \operatorname{put}_{\ell^{\prime}}(M, \Delta N)= & {\text { let } \Delta M^{\prime}=}^{\left(\Delta N^{+} \bowtie\{\{A=a\}\}, \Delta N^{-} \bowtie\{\{A=a\}\}\right) \text { in }} \\
& \left(\text { revise }_{X \rightarrow A}\left(\Delta M^{\prime+}, M\right), \text { revise }_{X \rightarrow A}\left(\Delta M^{\prime-}, M\right)\right)
\end{aligned}
$$

$\Delta M^{\prime}$ extends $\Delta N^{\prime}$ with the extra attribute $A$ set to the default value $a$, to match the domain of the underlying table. The final step optimises the use of drevise $_{X \rightarrow A}(\cdot, \cdot)$ in $\delta$ put $_{\ell}$ using Lemma 27 .

Theorem 4. [Correctness of optimised project lens] Suppose $M: \operatorname{Rel}(U, P, F)$ and $N=\pi_{U-A}(M)$. Suppose also that $\Delta N$ is minimal with respect to $N$ and that $N \oplus$ $\Delta N: \operatorname{Rel}\left(U-A, \pi_{U-A}(P), F^{\prime}\right)$, where $F \equiv F^{\prime} \uplus\{X \rightarrow A\}$. Then $\delta p u t_{\ell}(M, \Delta N)=$ $\delta p u t_{\ell^{\prime}}(M, \Delta N)$.

Proof. Shown in Appendix B. 2

### 3.2.4 Join Lens

The relational lens for the natural join operation comes with different update strategies that are all well-behaved. The put semantics for the join lens offers us some choice in where to delete records which should not appear in the output, but that can either be deleted from the left table or the right table. The simplest case is the variant that always tries to delete the record from the left table. The typing rule for the join lens have a condition on the functional dependencies that require the join key to completely define the domain of the right table. As a result, deleting an entry in the left table always only deletes a single record in the output. This means that deleting from the left table is always correct behaviour for removing a single record.

The incremental lens $\ell=\delta$ join_dl : $\operatorname{Rel}(U, P, F) \otimes \operatorname{Rel}(V, Q, G) \Leftrightarrow \operatorname{Rel}(U \cup V, P \bowtie$ $Q, F \cup G)$ is the lens join_dl defined in Figure 2.10 of the same type, equipped
with $\delta p u t_{\ell}$ defined as follows:

$$
\begin{aligned}
\text { oput }_{\ell}: & \operatorname{Rel}(U, P, F) \times \operatorname{Rel}(V, Q, G) \times \Delta \operatorname{Rel}(U \cup V, P \bowtie Q, F \cup G) \\
& \rightarrow \Delta \operatorname{Rel}(U, P, F) \times \Delta \operatorname{Rel}(V, Q, G) \\
\text { pput }_{\ell}((M, N), \Delta O)= & \operatorname{let} O=M \bowtie N \text { in } \\
& \operatorname{let}\left(M_{0}, \Delta M_{0}\right)=\operatorname{merge}_{F}^{\dagger}\left((M, \varnothing), \pi_{U}^{\dagger}(O, \Delta O)\right) \text { in } \\
& \operatorname{let}\left(N^{\prime}, \Delta N^{\prime}\right)=\operatorname{merge}_{G}^{\dagger}\left((N, \varnothing), \pi_{V}^{\dagger}(O, \Delta O)\right) \text { in } \\
& \operatorname{let}(L, \Delta L)=\left(\left(M_{0}, \Delta M_{0}\right) \bowtie^{\dagger}\left(N^{\prime}, \Delta N^{\prime}\right)\right)-^{\dagger}(O, \Delta O) \text { in } \\
& \operatorname{let} \Delta M^{\prime}=\left(M_{0}, \Delta M_{0}\right)-\pi_{U}^{\dagger}(L, \Delta L) \text { in } \\
& \left(\Delta M^{\prime}, \Delta N^{\prime}\right)
\end{aligned}
$$

We define $O$ as $p u t_{\text {join_dr }}(M, N)$. The tuples $\left(M_{0}, \Delta M_{0}\right)$ and $\left(N^{\prime}, \Delta N^{\prime}\right)$ compute the underlying tables $M$ and $N$ with all functional dependency changes applied. All records that must be deleted to ensure well-behavedness are computed as the tuple $(L, \Delta L)$, where $L$ is always the empty set and all records are contained in $\Delta L^{+}$. The delete left behaviour deletes all records from the left table, which is always permitted behaviour. The delta $\Delta M^{\prime}$ represents the changes necessary to $M$ and $\Delta N^{\prime}$ contains the changes to the right table.

Lemma 32. The incremental join lens jjoin_dl is well-behaved.

Proof. Follows from Theorem 2.
Definition 15. Define an optimised incremental $\delta j o i n-d l ~ l e n s ~ \ell^{\prime}$ with $\delta p u t_{\ell^{\prime}} d e-$ fined as follows:

$$
\begin{aligned}
\delta \text { put }_{\ell^{\prime}}((M, N), \Delta O)= & \text { let } P_{M}=\operatorname{affected}_{F^{* U}}\left(\pi_{U}\left(\Delta O^{+}\right)\right) \text {in } \\
& \text { let } P_{N}=\operatorname{affected}_{G^{* V}}\left(\pi_{V}\left(\Delta O^{+}\right)\right) \text {in } \\
& \text { let } \Delta M_{0}=\operatorname{merge}_{F}\left(\sigma_{P_{M}}(M), \pi_{U}\left(\Delta O^{+}\right)\right) \ominus \sigma_{P_{M}}(M) \text { in } \\
& \text { let } \Delta N^{\prime}=\operatorname{merge}_{G}\left(\sigma_{P_{N}}(N), \pi_{V}\left(\Delta O^{+}\right)\right) \ominus \sigma_{P_{N}}(N) \text { in } \\
& \text { let } \Delta L=\left(\left(\left(M \oplus \Delta M_{0}\right) \bowtie \Delta N^{\prime+}\right) \cup\left(\Delta M_{0}^{+} \bowtie\left(N \oplus \Delta N^{\prime}\right)\right),\right. \\
& \left.\quad\left(\Delta M_{0}^{-} \bowtie N\right) \cup\left(M \bowtie \Delta N^{\prime-}\right)\right) \ominus \Delta O \text { in } \\
& \text { let } \Delta M^{\prime}=\Delta M_{0} \ominus\left(\pi_{U}\left(\Delta L^{+}\right), \varnothing\right) \text { in } \\
& \left(\Delta M^{\prime}, \Delta N^{\prime}\right)
\end{aligned}
$$

In the optimised join lens, $\Delta M_{0}$ and $\Delta N^{\prime}$ can be calculated by first querying $\sigma_{P_{M}}(M)$ and $\sigma_{P_{N}}(N)$, where $P_{M}$ and $P_{N}$ include all rows potentially affected by merging functional dependencies, and then performing the appropriate relational
merges using $\pi_{U}\left(\Delta O^{+}\right)$and $\pi_{V}\left(\Delta O^{+}\right)$. Since this step may result in additional rows being generated or deleted rows not being removed, any excess rows $\Delta L$ are determined by calculating which rows would have been changed in the joined view after updates to the underlying tables $\Delta M_{0}$ and $\Delta N^{\prime}$, and then comparing those to the desired changes $\Delta O$. We can calculate $\Delta L$ efficiently by querying the underlying $M$ and $N$ tables only for records having identical join keys to records in $\Delta M_{0}$ and $\Delta N^{\prime}$.

Finally, the updated left table can be calculated as the changes to the left table $\Delta M_{0}$ minus all records that need to be removed to ensure the lens is well behaved. The changes $\Delta N^{\prime}$ are used for the right table.

## Join Template Lens

Rather than showing correctness for all join lens variants, we instead incrementalise the join template, which is the more general form of the join lens and can then be used to derive other join lens variants. The template join function takes two predicates $P_{d}$ and $Q_{d}$, where $P_{d}$ is a predicate from the record to a boolean indicating if the record should be deleted in the left table and $Q_{d}$ specifies if it should be deleted from the right table.

Consider a system which joins a collection of records with another table referencing the records. Here the system should retain records if they are either newer than 2018 or referenced by other rows. The deletion predicates to $P_{d}=$ date $<2018$ and $Q_{d}=\top$ satisfy the requirements. If the user deletes the last entry referencing a record and the record is older than 2018, then the record in the underlying records table is also deleted.

For the lens to be well-behaved, either $P_{d}$ or $Q_{d}$ must return true for any record. Section 5.2 shows this check can be performed statically. The join lens variants can all be derived by instantiating the join template lens with different $P_{d}$ and $Q_{d}$ values.

The incremental lens $\ell=\delta \operatorname{join}_{P_{d}, Q_{d}}: \operatorname{Rel}(U, P, F) \otimes \operatorname{Rel}(V, Q, G) \Leftrightarrow \operatorname{Rel}(U \cup V, P \bowtie$ $Q, F \cup G)$ is the lens $\operatorname{join}_{P_{d}, Q_{d}}$ of the same type, equipped with $\delta p u t_{\ell}$ defined as
follows:

$$
\begin{aligned}
\delta p u t_{\ell}: & \operatorname{Rel}(U, P, F) \times \operatorname{Rel}(V, Q, G) \times \Delta \operatorname{Rel}(U \cup V, P \bowtie Q, F \cup G) \\
& \rightarrow \Delta \operatorname{Rel}(U, P, F) \times \Delta \operatorname{Rel}(V, Q, G) \\
{\delta p u t_{\ell}}_{\ell}((M, N), \Delta O)= & \operatorname{let} O=M \bowtie N \text { in } \\
& \operatorname{let}\left(M_{0}, \Delta M_{0}\right)=\operatorname{merge}_{F}^{\dagger}\left((M, \varnothing), \pi_{U}^{\dagger}(O, \Delta O)\right) \text { in } \\
& \operatorname{let}\left(N_{0}, \Delta N_{0}\right)=\operatorname{merge}_{G}^{\dagger}\left((N, \varnothing), \pi_{V}^{\dagger}(O, \Delta O)\right) \text { in } \\
& \operatorname{let}(L, \Delta L)=\left(\left(M_{0}, \Delta M_{0}\right) \bowtie^{\dagger}\left(N_{0}, \Delta N_{0}\right)\right)-^{\dagger}(O, \Delta O) \text { in } \\
& \operatorname{let}\left(L_{l}, \Delta L_{l}\right)=(L, \Delta L) \bowtie^{\dagger} \pi_{U \cap V}^{\dagger}(O, \Delta O) \text { in } \\
& \operatorname{let}\left(L_{a}, \Delta L_{a}\right)=(L, \Delta L)-\dagger\left(L_{l}, \Delta L_{l}\right) \text { in } \\
& \text { let } \Delta M^{\prime}=\left(M_{0}, \Delta M_{0}\right)-\pi_{U}^{\dagger}\left(\left(L_{l}, \Delta L_{l}\right) \cup^{\dagger} \sigma_{P_{d}}^{\dagger}\left(L_{a}, \Delta L_{a}^{+}\right)\right) \text {in } \\
& \text { let } \Delta N^{\prime}=\left(N_{0}, \Delta N_{0}\right) \dot{-} \pi_{V}^{\dagger}\left(\sigma_{Q_{d}}^{\dagger}(L, \Delta L)\right) \text { in } \\
& \left(\Delta M^{\prime}, \Delta N^{\prime}\right)
\end{aligned}
$$

The join delete lens works by first restoring functional dependency consistency on the underlying tables using the relational merge function. The resulting $M_{0}$ and $N_{0}$ tables may introduce additional records in the output, which must be removed. These additional records are computed as $L$. Some of the records in $L$ can only be removed from the output by deleting the entry in the left table. Consider the join of the albums and tracks tables. If the entry for the song Lullaby with the album Galore should be removed from the output, it would not be correct to remove the Galore entry from the albums table as other entries in the joined view depend on it. Instead the corresponding entry in the tracks table must be removed. We compute $L_{l}$ as the records that must be deleted from the left table, and all remaining entries, computed as $L_{a}$, can be deleted from either tables.

Lemma 33. The incremental join lens $\delta \operatorname{join}_{P_{d}, Q_{d}}$ is well-behaved.
Proof. Follows from Theorem 2.

Definition 16. Define an optimised incremental $\delta^{\operatorname{join}}{ }_{P_{d}, Q_{d}}$ lens $\ell^{\prime}$ with $\delta p u t_{\ell^{\prime}}$
defined as follows:

$$
\begin{aligned}
\text { dput }_{\ell^{\prime}}((M, N), \Delta O)= & \text { let } P_{M}=\operatorname{affected}_{F^{* U}}\left(\pi_{U}\left(\Delta O^{+}\right)\right) \boldsymbol{i n} \\
& \text { let } P_{N}=\operatorname{affected}_{G^{* U}}\left(\pi_{V}\left(\Delta O^{+}\right)\right) \boldsymbol{i n} \\
& \text { let } \Delta M_{0}=\operatorname{merge}_{F}\left(\sigma_{P_{M}}(M), \pi_{U}\left(\Delta O^{+}\right)\right) \ominus \sigma_{P_{M}}(M) \boldsymbol{i n} \\
& \text { let } \Delta N_{0}=\operatorname{merge}_{G}\left(\sigma_{P_{N}}(N), \pi_{V}\left(\Delta O^{+}\right)\right) \ominus \sigma_{P_{N}}(N) \boldsymbol{i n} \\
& \text { let } \Delta L=\left(\left(\left(M \oplus \Delta M_{0}\right) \bowtie \Delta N_{0}+\right) \cup\left(\Delta M_{0}^{+} \bowtie\left(N \oplus \Delta N_{0}\right)\right),\right. \\
& \left.\quad\left(\Delta M_{0}{ }^{-} \bowtie N\right) \cup\left(M \bowtie \Delta N_{0}^{-}\right)\right) \ominus \Delta O \text { in } \\
& \text { let } \Delta L_{l}^{+}=\Delta L^{+} \bowtie \pi_{U \cap V}(O \oplus \Delta O) \text { in } \\
& \text { let } \Delta L_{a}^{+}=\Delta L^{+}-\Delta L_{l}^{+} \text {in } \\
& \text { let } \Delta M^{\prime}=\Delta M_{0} \ominus\left(\pi_{U}\left(\Delta L_{l}^{+} \cup \sigma_{P_{d}}\left(\Delta L_{a}^{+}\right)\right), \varnothing\right) \text { in } \\
& \text { let } \Delta N^{\prime}=\Delta N_{0} \ominus\left(\pi_{V}\left(\sigma_{Q_{d}}\left(\Delta L_{a}^{+}\right)\right), \varnothing\right) \boldsymbol{i n} \\
& \left(\Delta M^{\prime}, \Delta N^{\prime}\right)
\end{aligned}
$$

The template lens definitions for $\Delta M_{0}, \Delta N_{0}$ and $\Delta L$ are identical to those in the delete left lens. To calculate $\Delta L_{l}^{+}$, the program can query only those values in $O$ which have the same join value as those values in $\Delta L^{+}$. Calculating the remaining variables then becomes straightforward.

Theorem 5. [Correctness of optimised join lens] Suppose $M: \operatorname{Rel}(U, P, F)$ and $N: \operatorname{Rel}(V, Q, G)$ and $O=M \bowtie N$. Suppose also that $\Delta O$ is minimal with respect to $O$, and $O \oplus \Delta O: \operatorname{Rel}(U \cup V, P \bowtie Q, F \cup G)$. Then $\delta p u t_{\ell}((M, N), \Delta O)=$ $\delta p_{\ell^{\prime}}((M, N), \Delta O)$.

Proof. Shown in Appendix B.2.
Using the template we can show that the semantics for the $\delta$ join_dl lens are equivalent to the semantics of a $\delta$ join $_{\mathrm{T}, \perp}$ lens.

Lemma 34. Suppose $\ell=\delta^{\operatorname{join}}{ }_{T, \perp}$ and $\ell^{\prime}=\delta$ join_dl . Then $\delta p u t_{\ell}=\delta$ put $_{\ell^{\prime}}$.

Proof.
$P_{d}=\mathrm{T}$
suppose $P_{d}$
$Q_{d}=\perp$
suppose $Q_{d}$
$O: U \cup V, \Delta O: \Delta(U \cup V)$
suppose $O, \Delta O$
$\Delta M_{0}: U$
suppose $\Delta M_{0}$
$\Delta N_{0}: V$

$$
\begin{array}{ll}
\Delta L^{+}: U \cup V & \text { suppose } \Delta L^{+} \\
\Delta L_{l}^{+}=\Delta L^{+} \bowtie \pi_{U \cap V}(O \oplus \Delta O) & \text { suppose } \Delta L_{l}^{+} \\
\Delta L_{a}^{+}=\Delta L^{+}-\Delta L_{l}^{+} & \text {suppose } \Delta L_{a}^{+} \\
\Delta M^{\prime}=\Delta M_{0} \ominus\left(\pi_{U}\left(\Delta L_{l}^{+} \cup \sigma_{P_{d}}\left(\Delta L_{a}^{+}\right)\right), \varnothing\right) & \text { suppose } \Delta M^{\prime} \\
\Delta N^{\prime}=\Delta N_{0} \ominus\left(\pi_{U}\left(\sigma_{Q_{d}}\left(\Delta L_{a}^{+}\right)\right)\right. & \text {suppose } \Delta N^{\prime}
\end{array}
$$

We first show that $\Delta L_{l}^{+}$is a subset of $\Delta L^{+}$:
$\Delta L_{l}^{+}$

$$
\begin{aligned}
& =\Delta L^{+} \bowtie \pi_{U \cap V}(O \oplus \Delta O) \\
& =\pi_{U \cup V}\left(\Delta L^{+} \bowtie \pi_{U \cap V}(O \oplus \Delta O)\right) \\
\Rightarrow & \Delta L_{L}^{+} \subseteq \Delta L^{+}
\end{aligned}
$$

suppose $\Delta L_{l}^{+}$ $\pi_{U \cup V}(\cdot)$ unit

This is used to show that $\Delta L_{l}^{+} \cap \Delta L^{+}$is equal to $\Delta L_{l}^{+}$:
$\Delta L_{l}^{+} \cap \Delta L^{+}=\Delta L_{l}^{+} \quad \Delta L_{l}^{+} \subseteq \Delta L^{+} ; \cap$ induce $\subseteq$
The records deleted from the left and right table are equal to $\Delta L^{+}$:

$$
\begin{align*}
& \Delta L_{l}^{+} \cup \Delta L_{a}^{+} \\
& \quad=\Delta L_{l}^{+} \cup\left(\Delta L^{+}-\Delta L_{l}^{+}\right) \\
& \quad=\left(\Delta L_{l}^{+} \cap \Delta L^{+}\right) \cup\left(\Delta L^{+}-\Delta L_{l}^{+}\right)  \tag{1}\\
& \\
& \quad=\Delta L^{+}
\end{align*}
$$

$$
=\Delta L_{l}^{+} \cup\left(\Delta L^{+}-\Delta L_{l}^{+}\right) \quad \text { def. } \Delta L_{l}^{+}
$$

$$
\Delta L_{l}^{+}=\Delta L_{l}^{+} \cap \Delta L^{+}
$$

$$
\text { - and } \cap \text { complementary }
$$

We now show that the $\Delta M^{\prime}$ and $\Delta N^{\prime}$ expressions match:
$\Delta M^{\prime}$

$$
\begin{align*}
& =\Delta M_{0} \ominus\left(\pi_{U}\left(\Delta L_{l}^{+} \cup \sigma_{P_{d}}\left(\Delta L_{a}^{+}\right)\right), \varnothing\right) \\
& =\Delta M_{0} \ominus\left(\pi_{U}\left(\Delta L_{l}^{+} \cup \sigma_{T}\left(\Delta L_{a}^{+}\right)\right), \varnothing\right) \\
& =\Delta M_{0} \ominus\left(\pi_{U}\left(\Delta L_{l}^{+} \cup \Delta L_{a}^{+}\right), \varnothing\right) \\
& =\Delta M_{0} \ominus\left(\pi_{U}\left(\Delta L^{+}\right), \varnothing\right) \tag{11}
\end{align*}
$$

def. $\Delta M^{\prime}$
def. $P_{d}$ def. $\sigma_{\top}(\cdot)$
$\Delta N^{\prime}$

$$
\begin{array}{lr}
=\Delta N_{0} \ominus\left(\pi_{U}\left(\sigma_{Q_{d}}\left(\Delta L_{a}^{+}\right)\right), \varnothing\right) & \text { def. } \Delta N^{\prime} \\
=\Delta N_{0} \ominus\left(\pi_{U}\left(\sigma_{\perp}\left(\Delta L_{a}^{+}\right)\right), \varnothing\right) & \text { def. } Q_{d} \\
=\Delta N_{0} \ominus\left(\pi_{U}(\varnothing), \varnothing\right) & \sigma_{\perp}(\cdot)=\varnothing \\
=\Delta N_{0} \ominus(\varnothing, \varnothing) & \pi_{U}(\varnothing)=\varnothing
\end{array}
$$

## Join Delete Right

The template lens allows other lens variants to be easily derived. The optimised join lens with delete right semantics is derived from the template lens with $P_{d}=\perp$ and $Q_{d}=\mathrm{T}$ :

Definition 17. Define an optimised incremental jjoin_dr lens $\ell$ with $\delta p u t_{\ell} d e-$ fined as follows:

$$
\begin{aligned}
\delta p u t_{\ell}((M, N), \Delta O)= & \text { let } P_{M}=\operatorname{affected}_{F^{* U}}\left(\pi_{U}\left(\Delta O^{+}\right)\right) \text {in } \\
& \text { let } P_{N}=\operatorname{affected}_{G^{* V}}\left(\pi_{V}\left(\Delta O^{+}\right)\right) \text {in } \\
& \text { let } \Delta M_{0}=\operatorname{merge}_{F}\left(\sigma_{P_{M}}(M), \pi_{U}\left(\Delta O^{+}\right)\right) \ominus \sigma_{P_{M}}(M) \text { in } \\
& \text { let } \Delta N_{0}=\operatorname{merge}_{G}\left(\sigma_{P_{N}}(N), \pi_{V}\left(\Delta O^{+}\right)\right) \ominus \sigma_{P_{N}}(N) \text { in } \\
& \text { let } \Delta L=\left(\left(\left(M \oplus \Delta M_{0}\right) \bowtie \Delta N_{0}^{+}\right) \cup\left(\Delta M_{0}^{+} \bowtie\left(N \oplus \Delta N_{0}\right)\right),\right. \\
& \left.\left(\Delta M_{0}^{-} \bowtie N\right) \cup\left(M \bowtie \Delta N_{0}^{-}\right)\right) \ominus \Delta O \text { in } \\
& \text { let } \Delta L_{l}^{+}=\Delta L^{+} \bowtie \pi_{U \cap V}(O \oplus \Delta O) \text { in } \\
& \text { let } \Delta L_{a}^{+}=\Delta L^{+}-\Delta L_{l}^{+} \text {in } \\
& \text { let } \Delta M^{\prime}=\Delta M_{0} \ominus\left(\pi_{U}\left(\Delta L_{l}^{+}\right), \varnothing\right) \text { in } \\
& \text { let } \Delta N^{\prime}=\Delta N_{0} \ominus\left(\pi_{V}\left(\Delta L_{a}^{+}\right), \varnothing\right) \boldsymbol{i n} \\
& \left(\Delta M^{\prime}, \Delta N^{\prime}\right)
\end{aligned}
$$

## Join Delete Both

The join lens which deletes records from both tables can be derived by setting $P_{d}=Q_{d}=\mathrm{T}$.

Definition 18. Define an optimised incremental jjoin_both lens $\ell$ with $\delta p u t_{\ell}$ defined as follows:

$$
\begin{aligned}
\operatorname{sput}_{\ell}((M, N), \Delta O)= & \text { let } P_{M}=\operatorname{affected}_{F^{* U}}\left(\pi_{U}\left(\Delta O^{+}\right)\right) \text {in } \\
& \text { let } P_{N}=\operatorname{affected}_{G^{* U}}\left(\pi_{V}\left(\Delta O^{+}\right)\right) \text {in } \\
& \text { let } \Delta M_{0}=\operatorname{merge}_{F}\left(\sigma_{P_{M}}(M), \pi_{U}\left(\Delta O^{+}\right)\right) \ominus \sigma_{P_{M}}(M) \text { in } \\
& \text { let } \Delta N_{0}=\operatorname{merge}_{G}\left(\sigma_{P_{N}}(N), \pi_{V}\left(\Delta O^{+}\right)\right) \ominus \sigma_{P_{N}}(N) \text { in }
\end{aligned}
$$

$$
\begin{aligned}
& \text { let } \Delta L=\left(\left(\left(M \oplus \Delta M_{0}\right) \bowtie \Delta N_{0}^{+}\right) \cup\left(\Delta M_{0}^{+} \bowtie\left(N \oplus \Delta N_{0}\right)\right),\right. \\
&\left.\left(\Delta M_{0}^{-} \bowtie N\right) \cup\left(M \bowtie \Delta N_{0}^{-}\right)\right) \ominus \Delta O \text { in } \\
& \text { let } \Delta L_{l}^{+}= \Delta L^{+} \bowtie \pi_{U \cap V}(O \oplus \Delta O) \text { in } \\
& \text { let } \Delta L_{a}^{+}= \Delta L^{+}-\Delta L_{l}^{+} \text {in } \\
& \text { let } \Delta M^{\prime}= \Delta M_{0} \ominus\left(\pi_{U}\left(\Delta L^{+}\right), \varnothing\right) \text { in } \\
& \text { let } \Delta N^{\prime}=\Delta N_{0} \ominus\left(\pi_{V}\left(\Delta L_{a}^{+}\right), \varnothing\right) \text { in } \\
&\left(\Delta M^{\prime}, \Delta N^{\prime}\right)
\end{aligned}
$$

The definition for this join lens follows from Lemma 34 .
Corollary 4. Suppose $\ell=\delta j o i n_{-}$both and $\ell^{\prime}=\delta$ join $_{\top}, \mathrm{T}$. Then $\delta p u t_{\ell}=\delta p u t_{\ell^{\prime}}$.

### 3.2.5 Rename Lens

The incremental lens $\ell=\delta$ rename $_{A / B}: \operatorname{Rel}(U, P, F) \Leftrightarrow \operatorname{Rel}\left(U[A / B], \rho_{A / B}(P), F[A / B]\right)$, where $A \in U$ and $B \notin U$, is the lens rename ${ }_{A / B}$ with the additional function $\delta p u t_{\ell}$ defined as follows:

$$
\begin{aligned}
\delta p u t_{\ell}: & \operatorname{Rel}(U, P, F) \times \Delta \operatorname{Rel}\left(U[A / B], \rho_{A / B}(P), F[A / B]\right) \rightarrow \Delta \operatorname{Rel}(U, P, F) \\
{\delta p u t_{\ell}}_{\ell}(M, \Delta N)= & \operatorname{let} N=\rho_{B / A}(M) \text { in } \\
& \operatorname{let}\left(M^{\prime}, \Delta M^{\prime}\right)=\rho_{B / A}^{\dagger}(N, \Delta N) \text { in } \\
& \Delta M^{\prime}
\end{aligned}
$$

Definition 19. Define an optimised incremental $\delta$ rename $_{A / B}$ lens $\ell^{\prime}$ with $\Delta$ put $_{\ell^{\prime}}$ defined as follows:

$$
\operatorname{\delta put}_{\ell^{\prime}}(M, \Delta N)=\left(\rho_{B / A}\left(\Delta N^{+}\right), \rho_{B / A}\left(\Delta N^{-}\right)\right)
$$

The optimised rename lens performs the inverse rename operation on both components of the delta relation. No database queries are required.

Theorem 6. [Correctness of rename lens] Suppose $M: \operatorname{Rel}(U, P, F)$ and $N=$ $\rho_{A / B}(M)$, and that $\Delta N$ is minimal with respect to $N$ and satisfies $N \oplus \Delta N$ : $\operatorname{Rel}(U \cup V, P \bowtie Q, F \cup G)$. Then $\delta p u t_{\ell}(M, \Delta N)=\operatorname{sput}_{\ell^{\prime}}(M, \Delta N)$.

### 3.3 Evaluation

We have implemented both naive and incremental relational lenses in Links. Given the lack of an existing implementation of relational lenses, for the naive version we implemented the lenses described by Bohannon et al. [12]. A benefit is a fairer performance comparison, as the non-incremental relational lenses are implemented using the same set operation implementation as the incremental version. We evaluate the performance of the optimised $\delta p u t$ operations defined earlier.

Each performance experiment follows a similar pattern and are all executed on an AMD 3700X with 32GB of RAM and a solid state drive. The experiments were run in a virtual machine running Ubuntu 20.04.3 LTS and PostgreSQL 12 was used to host the database. Links was compiled using OCaml 4.12.0. All generated tables contained a primary key index and no other indices.

### 3.3.1 Microbenchmarks

## Lens Primitives

We first evaluate lens change-propagation performance as a set of microbenchmarks over the lens primitives, using two metrics: total time to compute the source delta for a single lens given a view delta as input, or in the case of the naive lenses, to calculate new source tables, referred to as total execution time. We also measure the amount of the total execution time which can be attributed to query execution.

The following steps are taken for each benchmark:

1. Generate the required tables with a specified set of columns and fill the tables with random data. For test purposes, the random data can either be sequential, a bounded random number or a random number. The microbenchmarks use the following tables:

- Table $t_{1}$ with domain $A, B$ and $C$ and functional dependency $A \rightarrow B C$. Populated with $n$ rows, with $A$ calculated as a sequential value, $B$ a random number up to $n / 10$ and $C$ a random number up to 100 .
- Table $t_{2}$ with domain $B$ and $D$ and functional dependency $B \rightarrow D$.

Populated with $n / 10$ rows with $B$ being a sequential value and $D$ a random number up to $n / 10$.
2. Generate lenses for the underlying tables and compose the required lenses on top of these. Lenses that use both $t_{1}$ and $t_{2}$ always start by joining the two tables on column $B$.
3. Fetch the output view of the lens using get and then make changes in a systematic fashion as described for each setup. The changes are designed to affect a small portion of the database. Most changes are of the form: "take all rows with attribute $A$ having some value, and update attribute B".
4. Apply put for the first lens using the updated view. For the incremental lenses this means first calculating the delta from the view (not timed) and then timing $\delta p u t$ which calculates the source delta. For naive lenses we measure the time required to recompute the source table using nonincremental put, but not the time needed to update the database. This process is repeated multiple times (ignoring the first few runs) and then taking the median value.

We repeat this process for each of the select, projection and join lenses. We exclude the time required to calculate the view delta or update the database because these operations only need to be performed once per lens, regardless of the number of intermediate steps defining the lens. Instead, we measure the performance of these one-time costs later in this Section, and present a complete example involving two lens primitives in Section 3.3.2.

We assume the following primitive lenses in the remaining benchmarks:

$$
\begin{aligned}
& \text { let } t_{1}=\text { lens } t 1 \text { of }(A: \operatorname{int}, B: \operatorname{int}, C: \text { int }) \text { with } A \rightarrow B C \\
& \text { let } t_{2}=\text { lens } t 2 \text { of }(B: \text { int }, D: \operatorname{int}) \text { with } B \rightarrow D
\end{aligned}
$$

Select example The select benchmark uses both $t_{1}$ and $t_{2}$. We compose a select lens on top of the join lens with the predicate $C=3$, which produces a view with an average size of $n / 100$ entries. The view is modified so that all records with $0 \leq B \leq 100$ have their $D$ value set to 5 . This approach produces deltas of length 20 on average, containing one row removal and one row addition. The lens used


Figure 3.5: Total and query execution time required by individual lens primitives vs. underlying table sizes
by the benchmark is as follows:

$$
\begin{aligned}
& \text { let } l=\text { join_dl } t_{1} \text { with } t_{2} \text { in } \\
& \operatorname{select}_{C=3} \text { from } l
\end{aligned}
$$

The naive put operation makes use of a single query and requires a total computation time of between 1 ms and 2919 ms depending on the row count. While the performance is acceptable for small tables, it is still too slow for most applications as the tables become larger. It also shows how an unavoidable bottleneck is introduced, as the query time reaches up to 615 ms .

Depending on the row count, the incremental version only needs between $<1 \mathrm{~ms}$ and 36 ms total computation time. Of this time, between $<1 \mathrm{~ms}$ and 35 ms are used to perform the required queries, accounting for the majority of the total computation time. The execution time and query time scale proportionally to the data size. The incremental performance reflects the fact that the view is much smaller than the entire table, which needs to be recomputed for the nonincremental version. It may be possible to improve performance by configuring the database to index $C$, since this may reduce the query execution time. Indexing would not affect naive performance, since we always fetch the complete source tables.

Projection performance We define a drop lens over the table $t_{1}$ removing attribute $C$, which is determined by the attribute $A$ and a default value of 1 . The view is modified by setting $B$ to 5 for all records where $60<A \leq 80$. This process modifies 20 of the $n$ records in the view. The lens used by the benchmark is:

$$
\text { drop } C \text { determined by }(A, 1) \text { from } t_{1}
$$

The performance of the lens is shown in Figure 3.5b. As in the case of the join lens, the naive projection lens implementation quickly becomes infeasible, requiring a total execution time of over 2499 ms as it processes 200000 records. The naive version spends up to $434 m s$ querying the server. The incremental version is able to perform the $\delta p u t$ operation in $<1 \mathrm{~ms}$ and less for the given row counts. This time includes the time required to query the database server for additional information.


Figure 3.6: Other join lens variants.

Join example The join benchmark uses the join lens defined over the two tables $t_{1}$ and $t_{2}$. We fetch the resulting view, which will contain $n$ rows. After that we modify all records containing a value for $B$ between 40 and 50 and set their $C$ value to 5 .

$$
\text { join_dl } t_{1} \text { with } t_{2}
$$

We benchmark the described setup with $n$ values ranging from 500 to 200000, timing the lens put duration for each $n$ as specified. The performance results are shown in Figure 3.5 c . The put operation for the naive join requires two queries but quickly becomes impractical. Of the computation time, approximately 1 ms to 641 ms depending on the table size is required for querying the database. While the query time taken by the naive approach is relatively low, this is due to the fact that the tables are relatively small and the time increases to hundreds of milliseconds as the table size grows to hundreds of thousands of rows.

In comparison, the incremental approach can scale to hundreds of thousands of rows and requires only 1 ms to 3 ms of both computation and query time for the given views. It requires 4 queries which are all simple to compute and return small views.

Figure $\sqrt{3.5 \mathrm{c}}$ shows the performance of the join delete left variant. If different deletion behaviour is required, then at least 5 queries are required. Figure 3.6 shows benchmarks for the other lens variants. These perform similarly to the delete left variant, requiring up to $5414 m s$ for the naive lens, of which up to 658 ms is required to query the database server. The incremental version only required up to $4 m s$ of which up to $2 m s$ is required to query the server.

Summary The above experiments show that incremental evaluation outperforms naive evaluation of relational lenses. We summarise the performance of incremental tables in Table 3.1, which shows the number of queries, query evaluation time and total evaluation time for all three microbenchmarks discussed above.

|  | select | drop | join delete left | join delete both | join delete right |
| :---: | :---: | :---: | :---: | :---: | :---: |
| query count | 1 | 1 | 4 | 5 | 5 |
| query $n=200 k$ | $30 m s$ | $<1 m s$ | $1 m s$ | $1 m s$ | $1 m s$ |
| total $n=200 k$ | $30 m s$ | $<1 m s$ | $2 m s$ | $3 m s$ | $2 m s$ |

Table 3.1: Query counts and times for large data sizes

## Delta Calculation Performance

While microbenchmarks on lens primitives give us some insight into the performance of the lenses, they do not account for the time required to calculate the initial delta, which is only required for incremental lenses. We modify the view of the join lens defined over $t_{1}$ and $t_{2}$ by fetching the view using get, and by then performing changes as done in the other experiments. Specifically we set $B$ to 5 for all records where $0<D<10$. Given that this example does not have any selection lenses, the size of the view will always be $n$.

We measure the time taken to fetch the original view from the database and then subtract it from a modified view. As in the previous examples we measure both the time required to query the database server as well as the total execution time on the client. We measure the time required for $n$ values ranging between 100 and 200000. The results are shown in Figure 3.7a,

Both the query and execution time are approximately linear with respect to the number of input rows. We require between $<1 \mathrm{~ms}$ and 2615 ms to compute the delta, of which 609 ms for 200000 rows is spent querying the database.

## Delta Application Performance

We also measure the time it takes to apply a delta to a table. This process requires the generation of insert, update and delete SQL commands which must then be executed on the server.

(a) Time needed to compute a delta for any view depending on the row count.

(b) Time needed to apply a delta to the database depending on the number of changes.

(c) Delta propagation time as a function of view update size $(\mathrm{n}=100000)$

Figure 3.7: Evaluation of delta calculation, delta application and delta propagation time as a function of view update size.

We consider delta application for a single table. We use the table $t_{1}$ from Section 3.3.1 and populate it with $n=10000$. We generate a delta containing $m$ entries, where a quarter of the entries produce $m / 4$ insertions, another quarter produce $m / 4$ deletions and the remaining half produce $m / 2$ updates.

Given such a delta, we time how long it takes to produce the SQL commands from the already calculated delta with varying size $m$. The SQL update commands are concatenated and sent to the database together as a single transaction. As in the other cases we time both the total and query execution times, which are shown in Figure 3.7b, For the naive version, we generate an update command that deletes the current contents of the table and inserts the new contents. For the incremental version, we generate updates that insert, delete, or replace only affected records.

The figure shows that the naive version's performance is independent of the number of changes, requiring around 156 ms , most of which is spent querying the database. The incremental version, on the other hand, requires only 20 ms for 1000 changes, and linearly scales until it requires the same time as the naive version for 10000 rows.

## Varying Delta Size

In addition to varying the size of the underlying database tables we also consider how the size of the delta may affect the performance of an update. To do this we use the two tables $t_{1}$ and $t_{2}$ with $n=100000$ and define a select lens on top of the join lens with the predicate $C=3$. We then determine a $b^{\prime}$, starting from 0 in steps of 100 , so that modifying all records where $0<B<b^{\prime}$ by setting $D=5$ produces a delta of size greater than $m$.

As in the other microbenchmarks we measure the total and query execution time taken to perform the $\delta p u t$ of an already calculated view delta or, in the naive case, the time to recalculate the full source tables using put. We repeat this experiment for varying $m$ values, ranging from 10 to 1000. The resulting execution times are plotted in Figure 3.7c.

As would be expected, the naive lens is relatively constant, requiring around 410 ms regardless of the number of entries in the delta. The incremental version starts at 31 ms and slows down to 217 ms as the delta contains more entries. About
$80 \%$ of the incremental time is used to query the database. The incremental method is notably more efficient for a small number of changes.

### 3.3.2 DBLP Example

In addition to the microbenchmarks we also perform some experiments on a real-world example involving the DBLP Computer Science Bibliography [63], a comprehensive collection of bibliographic information about computer science publications. It is published as a large and freely available XML file with millions of records containing publications, conference proceedings, journals, authors, websites and more.

| inproceedings |
| :---: |
| 'conf/pods/BohannonPV06' |
| $\vdots$ |
| 'Relational lenses: |
| with $\{$ inproceedings $\rightarrow$ title year |
| proceedings $\}$ |

(a) The inproceedings table.

| inproceedings | author |
| :---: | :---: | :---: |
| 'conf/pods/BohannonPV06' | 'Aaron Bohannon' |
| 'conf/pods/BohannonPV06' | 'Benjamin C. Pierce' |
| 'conf/pods/BohannonPV06' | 'Jeffrey A. Vaughan' |
| $\vdots$ |  |
| with $\}$ |  |

(b) The inproceedings_author table.

Figure 3.8: The tables used in our DBLP example.

Our example uses a table containing a collection of conference publications called inproceedings as well as a table of their respective authors inproceedings_author. We use a parser to convert the given XML file into a set of PostgreSQL tables. This first table contains the title of the paper, the year it was published as well as the proceedings it is in, while the inproceedings_author table contains an entry for each author on every paper, allowing a single publication to have multiple authors. The tables are shown in Figure 3.8.


Figure 3.9: The functional dependencies in tree form of the DBLP example.

The join of the inproceedings and inproceedings_author tables produces a view with the columns author, inproceedings, title, year, proceedings. The functional dependencies of the joined table are shown in Figure 3.9.

In order to determine how the application scales for varying database sizes, we generate the underlying tables by selecting a set of entries so that the join of inproceedings and inproceedings_author contains $n$ rows. Given that set of entries, we select all entries in the complete inproceedings_author table, which have a corresponding entry in the subset of entries chosen.

Using these tables we join the two tables on the inproceedings attribute and then select all entries from PODS 2006. The Links code used to generate the lenses is shown below.

```
var joinL = lensjoin inproceedings_authorL with inproceedingsL on inproceedings;
var selectL = lensselect from joinL where proceedings == "conf/pods/2006";
```

Retrieving the view of the lens using get results in table containing the entries as shown below. We retrieve the view in links and make a change in the title to all entries with attribute inproceedings $=$ 'conf/pods/BohannonPV06'. The updated view is applied to the database using the put operation.

| author | inproceedings | title | year | proceedings |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 'Aaron Bohannon' | 'conf/pods/BohannonPV06' | 'Rel...' | 2006 | 'conf/pods/2006' |
| 'Benjamin C. Pierce' | 'conf/pods/BohannonPV06' | 'Rel...' | 2006 | 'conf/pods/2006' |
| 'Jeffrey A. Vaughan' | 'conf/pods/BohannonPV06' | 'Rel...' | 2006 | 'conf/pods/2006' |
| $\vdots$ |  |  |  |  |

As in the other examples we fetch the output of the select lens using get and then make the small modification. We then perform the put operation to apply those changes to the database. During the put we time the entire process of generating a delta from the view, calculating the delta for the underlying tables and updating the database for both the naive and incremental lenses. While timing we keep


Figure 3.10: The total/query execution time for the put operation applied to our DBLP database.
track of how much time was spent querying the database and the total time spent performing the operation. We also perform put using a database located on a remote server and compare it to a database located on the same machine.

Our performance results for the database hosted on the local machine are shown in Figure 3.10a. Similar to our earlier benchmarks, the incremental lenses perform favourably in comparison to the naive lenses. The naive lenses require linear time as the data grows and need up to 1085 ms to update the database when the data grows to 20000 rows. A large portion of the naive execution time (up to 497 ms ) is used to query the entire database, and so optimising the local algorithms will have limited effect on the overall performance.

In comparison, the incremental lenses perform much faster even as the data grows to hundreds of thousands of rows, requiring only between 7 ms and 10 ms total execution time depending on the size of the underlying tables. Of that time between 6 ms and 9 ms are used to query the database, making database performance the limiting factor for small data sets.

Figure 3.10b shows the performance of the same application running on a remote database server. Here the query time is increased due to the bandwidth limitation and higher latency of the network. Less bandwidth means that the data loads more slowly, while higher latency imposes a delay per query. The total execution time of the naive version increases by a significant amount up to 1438 ms , while the incremental version remains much faster requiring only up to 23 ms , of which 21 ms are used for querying the database.

Figure 3.10 c directly compares the performance of incremental relational lenses when the database is located on the same machine and a remote server. For small data sets the remote server introduces an overhead of around 9 ms . The overhead decreases as the data sets become larger, with the remote server being 5 ms faster than the local version for the largest data set. The remote database server is run on a faster computer which is able to execute queries faster than the local machine. Figure 3.10 c shows the performance of the application with data sets up to 200000 rows, indicating that the incremental version is able to easily scale up to larger data sets.

### 3.4 Summary

Relational lenses, as defined by Bohannon et al. [12], propagate changes to the view by completely recalculating the underlying tables. This approach is impractical because such databases may become very large and then each update to the database becomes very inefficient. It is also unsuitable for an implementation of language integrated relational lenses, where the application is separated from the database server.

Rather than recomputing entire sets of source tables, our incremental relational lenses only process the changes required. When the user submits an updated view, a delta between the updated view and the unchanged view is first computed. The delta is propagated down to the underlying database and can be applied by only altering the existing tables. During propagation, the lenses can query the database for additional information as required, preventing each materialized view from being instantiated.

The incremental semantics are derived by first introducing incremental relational algebra semantics, based on existing work by Griffin et al. [42]. This framework is then used to differentiate the put operation for each lens primitive. The resulting expression is then optimized to improve performance by simplifying the expression.

The incremental semantics of relational lenses are used by both the Links and Haskell implementations discussed in the remaining chapters of this thesis. We perform an experimental evaluation of the incremental semantics and compare the efficiency to the naive approach which recomputes the tables in memory and then replaces the entire table in the database. The evaluation demonstrates the ability of relational lenses to scale to databases with hundreds of thousands of rows, and show that the incremental semantics can perform the same computation in a fraction of the time.

Our work establishes for the first time the feasibility of relational lenses for solving classical view update problems in databases. Nevertheless, there may be room for improvement in various directions. We found a pragmatic solution that uses a small number of simple queries, but other strategies for calculating minimal deltas are possible. Developing additional incremental relational lens primitives or combinators, and combining relational lenses with conventional lenses, are two
other possible future directions.
This chapter concludes the changes to the relational lens semantics in this thesis. The following chapters look at how the relational lens typing rules can be implemented.

## Chapter 4

## Turning Abstract Sets into Concrete Predicates

In their original proposal for relational lenses, Bohannon et al. [12] define predicates using (potentially infinite) abstract sets. Although theoretically convenient, such a representation is not suited to implementation in a programming language. This chapter looks at how predicates can be represented, how the required checks can be performed on them and how they can be integrated into the target language.

As we are working in the setting of a programming language, it is natural to treat predicates as a function from a record to a binary value. In this case we treat all the fields of the record as bound in the environment. Recall our earlier example of the select lens, which selects albums with a given name:

$$
\text { select }_{\text {album }}=\text { "galore" } \text { from albums }
$$

Here, the predicate function checks that the album column of each record is equal to the constant value Galore. Intuitively, this predicate includes a record in the set of results if its album field matches Galore.

A challenge with implementing predicates for relational lenses is that some lenses require additional checks on predicates. Recall that the drop lens allows a more fine-grained notion of relational projection, allowing us to remove a column from a view. Consider a lens $l_{1}$ as a select lens on the tracks table with predicate date > $1990 \vee$ rating > 4.

| track | date | rating | album |
| :---: | :---: | :---: | :---: |
| Lovesong | 1989 | 5 | Galore |
| Lovesong | 1989 | 5 | Paris |
| Trust | 1992 | 4 | Wish |

We can then define the lens $l_{2}$ as dropping the column date determined by track from the lens $l_{1}$. This lens yields the following table, on which we would like to perform the updates marked in red:

| track | rating | album |  |
| :---: | :---: | :---: | :---: |
| Lovesong | $\boxed{y}$ | 3 | Galore |
| Lovesong | $\boxed{5}$ | 3 | Paris |
| Trust | 4 |  | Wish |

What would the new predicate constraint be? It cannot reference the field date, since it does not exist anymore. If it were rating $>4$ then the last record would be a violation in the output view. If the predicate were true it would violate PutGet: Changing the rating from 5 to 3 for the track Lovesong, would cause it to no longer satisfy the parent lens' predicate since it is from 1989 and the rating is only 3 .

For this example there is no valid predicate and the lens would not be wellbehaved. The lens should be rejected by the drop lens rule. Bohannon et al. [12] provide the necessary checks on abstract sets. This chapter looks at how such issues can be detected and thus prevented when working with our concrete predicate syntax.

The underlying issue is the dependency between the dropped field date and the field rating. It is not possible to define a predicate $P$ which specifies if any rating value is valid independently of the drop column date. Without being able to construct such a $P$, a lens cannot be well-behaved and should be rejected.

The first contribution of this chapter is the introduction of a concrete syntax for defining predicates. It is then shown how the predicate checks defined by Bohannon et al. [12] can be implemented on the concrete predicate syntax. The predicate checks on concrete predicates are proven identical to the equivalent checks on predicate sets.

The contributions described in this chapter are as follows:

- Section 4.1 introduces a concrete predicate language, which is easily translatable into SQL.
- Section 4.2 defines the checks required on predicates for our predicate language. These are proven correct w.r.t. the checks defined by Bohannon et al. [12].
- The basic syntax is then extended with further features not directly supported by SQL in Section 4.3 .
- We consider two language integration strategies in Section 4.4. The first strategy allows the predicate to be defined in the native language syntax, and covers the trade-offs between requiring predicates to be known statically during compilation and being more flexible but also performing the predicate checks during runtime. The other approach using hybrid predicates uses a more advanced language integration technique. This technique allows more flexibility while still providing compile-time guarantees.


### 4.1 Basic Predicates

Syntax We start with the basic predicate syntax shown in Figure 4.1. The predicate can either be a constant value $c$, a column reference $\ell$, a builtin operator application $\odot\{\vec{P}\}$ or conditional branching if $P$ then $Q_{1}$ else $Q_{2}$. Predicates do not allow any more advanced features, in particular function abstractions. Function applications only supports builtin operators $\odot$ and require a complete set of arguments, partial application is not allowed.

Typing This language only supports expressions that compute basic value types $A$ such as int. Complex types such as tuples and sets are not supported. As the basic language does not support lambda abstractions and function types, primitive operators receive special types $A_{1} \times \ldots \times A_{n} \rightarrow A$ which are function abstractions taking $n$ arguments of some type $A_{i}$ for $1 \leq i \leq n$, yielding a value of type $A$. These special function types are not considered part of the type system and so a primitive operator is not a well-typed expression by itself.

The typing rule judgement $R \vdash P: A$ ensures that the predicate expression $P$ has type $A$ when a record of type $R$ is bound to the environment. The typing rules

Syntax

| Labels | $\ell$ |  |
| :--- | ---: | :--- |
| Predicates | $P, Q$ | $::=c\|\ell\| \odot\{\vec{P}\} \mid$ if $P$ then $Q_{1}$ else $Q_{2}$ |
| Base Types | $A::=$ bool $\mid$ int $\mid$ string |  |

Typing rules
$R \vdash P: A$

|  | T-IF |  |
| :--- | :--- | :---: |
| T-VAR | T-Const | $R \vdash P:$ bool |
| $\ell: A \in R$ |  |  |
| $R \vdash \ell: A$ | $c$ of type $A$ |  |
| $R \vdash c: A$ |  | $R \vdash Q_{1}: A \quad R \vdash Q_{2}: A$ |
| $R \vdash$ if $P$ then $Q_{1}$ else $Q_{2}: A$ |  |  |

$$
\begin{aligned}
& \text { T-OP } \\
& \frac{\odot: A_{1} \times \ldots \times A_{n} \rightarrow A \quad\left(R \vdash P_{i}: A_{i}\right)_{i \in 1 . . n}}{R \vdash \odot\{\vec{P}\}: A}
\end{aligned}
$$

Evaluation rules
E-VAR
E-Const
E-Op
$\frac{(\ell=v) \in r}{\ell \Downarrow_{r} v} \quad \overline{c \Downarrow_{r} c}$
$\frac{\left(P_{i} \Downarrow_{r} v_{i}\right)_{i \in 1 . . n} \quad v=\odot\left\{v_{1}, \ldots, v_{n}\right\}}{\odot\{\vec{P}\} \Downarrow_{r} v}$
E-If-True
E-If-False
$\frac{P \Downarrow_{r} \text { true } \quad Q_{1} \Downarrow_{r} v}{\text { if } P \text { then } Q_{1} \text { else } Q_{2} \Downarrow_{r} v}$
$\frac{P \Downarrow_{r} \text { false } \quad Q_{2} \Downarrow_{r} v}{\text { if } P \text { then } Q_{1} \text { else } Q_{2} \Downarrow_{r} v}$

Figure 4.1: Syntax and typing rules for the basic predicate language.
shown in Figure 4.1 are standard typing rules. Labels $\ell$ are looked up in the environment $R$. Constant value expressions $c$ type to the primitive type of the constant. Conditional if expressions yield a type $A$ if the condition has type bool, and both of the branches type to $A$. Finally, any applications of builtin primitives should ensure all the arguments are supplied and have the correct argument type and will yield the result type of the primitive operator. The notation $v: A$ is shorthand notation for $\cdot \vdash v: A$, stating that the value $v$ is well-typed under an empty context.

Evaluation Figure 4.1 also introduces the big-step evaluation relation $P \Downarrow_{r} v$, which states that term $P$ evaluates to a value $v$ under the evaluation context $r$. We use the notation $\hat{\odot}\{\vec{v}\}$ to describe the denotation of operation $\odot$ applied to arguments $\vec{v}$ : for example, $\hat{+}\{5,10\}=15$. The semantics enjoys a standard type soundness property.

Proposition 7 (Type Soundness). If $R \vdash P: A$, then for any $r: R$ there exists some $v$ such that $P \Downarrow_{r} v$ and $v: A$.

Proof. By induction on $R \vdash P: A$.

The advantage of this basic predicate syntax is that it is straightforward to convert any predicate $P$ to SQL. The conversion of a predicate to SQL only requires syntactic changes such as converting primitive operations into their SQL equivalent (e.g. $\wedge$ becomes AND). While this language can express a lot of predicates, it has limited abstraction capabilities due to the limited types available and lack of functional abstractions.

A useful property to have for any predicate is the ability to extend the environment with further unreferenced variables. This lemma is also required for hybrid predicates in Section 4.4.1.

Lemma 35 (Predicate Weakening). Given a predicate $P$, if $R_{1} \vdash P: A$ then for any $R_{2}$ disjoint from $R_{1}$ it can be shown that $R_{1} \oplus R_{2} \vdash P: A$.

Proof. By induction on $R \vdash P: A$.

### 4.2 Predicate Checks

The checks required by relational lenses are one of the main challenges of providing relational lenses as a library. In this section we look at how the checks defined by Bohannon et al. [12] can be performed on our concrete predicate syntax.

We first look at the dependence between columns and how the lossless join decomposition check can be performed when working with our predicate syntax. This work reveals how the compatibility of the default value for the dropped predicate column can be ensured using the default value check. Finally, we look into how the dependence of the dropped column can be removed from the predicate.

### 4.2.1 Set Equivalence

We would like to show that the checks performed on predicates are sound. This requires us to show how predicates relate to set predicates. We begin with some preliminary definitions.

Definition 20 (Predicate satisfaction). We say that a record $r$ satisfies predicate $P$, written $\operatorname{sat}(P, r)$, if $P \Downarrow_{r}$ true.

Given a record type $R$, we define $\operatorname{inh}(R)$ as the set of all closed records $r$ of the given record type $R . \operatorname{inh}(R)$ is formally defined as:

Definition 21 (Record type inhabitants). We define the inhabitants of a record type $R$, written inh $(R)$, as:

$$
\{r \mid r: R\}
$$

We can then $\operatorname{define} \operatorname{set}(P, R)$ as the equivalent set of all records of type $R$ satisfying a predicate $P$. The definition of $\operatorname{set}(P, R)$ is used to show that our implementation is sound.

Definition 22 (Predicate sets). We define the set representation of predicate $P$ over $R$, written $\operatorname{set}(P, R)$, as:

$$
\{r \in \operatorname{inh}(R) \mid \operatorname{sat}(P, r)\}
$$

Lemma 36. $r \in \operatorname{set}(P, R)$ if and only if $\operatorname{sat}(P, r)$.

Proof. By definition of $\operatorname{sat}(P, \cdot), \cdot \in \cdot$ and $\operatorname{set}(\cdot, \cdot)$.

Lemma 37. Suppose two disjoint type contexts $R$ and $R^{\prime}$ as well as $r \in \operatorname{inh}(R)$ and $s \in \operatorname{inh}\left(R^{\prime}\right)$. Then $\operatorname{sat}(P, r \otimes s)$ implies $r \in\left\{z[R] \mid z \in \operatorname{inh}\left(R \oplus R^{\prime}\right)\right.$. sat $\left.(P, z)\right\}$ and $s \in\left\{z\left[R^{\prime}\right] \mid z \in \operatorname{inh}\left(R \oplus R^{\prime}\right)\right.$. $\left.\operatorname{sat}(P, z)\right\}$.

Proof. $\operatorname{sat}\left(P, r_{1} \otimes r_{2}\right)$ implies that there exists an $s \in \operatorname{inh}\left(R \oplus R^{\prime}\right)$ which is equal to $r_{1} \otimes r_{2}$ such that $\operatorname{sat}(P, s)$. We know that for $s$ it follows that $s\left[\operatorname{dom}\left(R_{1}\right)\right]$ equals $r_{1}$ by definition of $\cdot \otimes \cdot$. Conversely the same can be shown for $r_{2}$.

Lemma 38. Suppose two predicates $P, Q$ such that $R \vdash P$ and $R \vdash Q$. Then $\operatorname{set}(P \wedge Q, R)=\operatorname{set}(P, R) \cap \operatorname{set}(Q, R)$.

Proof.
$R \vdash P$ assumption
$R \vdash Q$
assumption

```
\(\operatorname{set}(P \wedge Q, R)\)
    \(=\{r \mid \forall r \in \operatorname{inh}(R) . \operatorname{sat}(P \wedge Q, r)\} \quad \operatorname{def.} \operatorname{set}(\cdot, \cdot)\)
    \(=\{r \mid \forall r \in \operatorname{inh}(R) \cdot \operatorname{sat}(P, r)\) and \(\operatorname{sat}(Q, r)\} \quad\) def. \(\cdot \wedge\) •
    \(=\{r \mid \forall r \in \operatorname{inh}(R) . r \in \operatorname{set}(P, R)\) and \(r \in \operatorname{set}(Q, R)\} \quad\) Lemma 36
    \(=\operatorname{set}(P, R) \cap \operatorname{set}(Q, R)\) def. \(\cdot \cap\).
```

Lemma 39. Suppose two predicates $P, Q$ such that $x: R \vdash P$ and $x: R^{\prime} \vdash Q$. Then $\operatorname{set}\left(P \wedge Q, R \oplus R^{\prime}\right)=\operatorname{set}(P, R) \bowtie \operatorname{set}\left(Q, R^{\prime}\right)$.

Proof.
$x: R \vdash P$
assumption (1)
$x: R^{\prime} \vdash Q$
assumption (2)
$\operatorname{set}\left(P \wedge Q, R \oplus R^{\prime}\right)$

$$
\begin{array}{lr}
=\left\{r \mid \forall r \in \operatorname{inh}\left(R \cup R^{\prime}\right) \cdot \operatorname{sat}(P \wedge Q, r)\right\} & \operatorname{def.} \operatorname{set}(\cdot, \cdot) \\
=\left\{r \mid \forall r \in \operatorname{inh}\left(R \oplus R^{\prime}\right) \cdot \operatorname{sat}(P, r) \text { and } \operatorname{sat}(Q, r)\right\} & \operatorname{def} \cdot \cdot \wedge \tag{12}
\end{array}
$$

$=\left\{r \mid \forall r \in \operatorname{inh}\left(R \oplus R^{\prime}\right) \cdot r[\operatorname{dom}(R)] \in \operatorname{set}(P, R)\right.$
and $\left.r\left[\operatorname{dom}\left(R^{\prime}\right)\right] \in \operatorname{set}\left(Q, R^{\prime}\right)\right\}$
Lemma 36
$=\operatorname{set}(P, R) \bowtie \operatorname{set}\left(Q, R^{\prime}\right) \quad \operatorname{def} . \bowtie \bowtie$.

### 4.2.2 Lossless Join Decomposition

For the resulting view table in the example from the introduction, a date value is attached to each record. The date value chosen is either some value that appears in the underlying table or it is the default value specified by the programmer. It must therefore be possible to attach any date value that could potentially appear in the underlying view to any valid record added to this view. Any added date value added to the resulting record should not violate the predicate $P$.

Assume a lens with domain $R$ equal to the disjoint union of $R_{1}$ and $R_{2}$, written $R_{1} \oplus R_{2}$, where $R_{2}$ should be dropped. The type checker should ensure that the lens predicate $P$ does not have a dependency between the dropped and remaining columns $R_{1}$ and $R_{2}$ that could cause issues as described. Bohannon et al. [12] perform this check by ensuring that the predicate set equivalent $M$ forms a lossless join decomposition.

This check ensures that there is a set of valid dropped column values equal to the projection of $M$ onto the columns of $R_{2}$, written $M\left[R_{2}\right]$, as well as a set of valid remaining column values equal to $M\left[R_{1}\right]$, such that any valid record $r \in M$ can have either its $R_{1}$ or $R_{2}$ component be exchanged by another valid value without changing the state of the record in the predicate set $M$. The check is performed by requiring that the original set of $M$ can be recovered by the natural join $\bowtie$ of the two sets $M\left[R_{1}\right]$ and $M\left[R_{2}\right]$.

A lossless join decomposition on sets is formally defined as follows:
Definition 23 (Lossless join decomposition). A set $M: R$ is $a$ lossless join decomposition of two record types $R_{1} \subseteq R$ and $R_{2} \subseteq R$ when:

$$
M=M\left[R_{1}\right] \bowtie M\left[R_{2}\right]
$$

The lossless join decomposition check, which is defined in set semantics, needs to be performed on predicates in our basic predicate syntax. To check the safety of a drop lens, we need to show that the predicate does not impose any dependency between the value of the dropped field and any other field. We formalise this constraint by defining the notion of a lossless join decomposition for predicate expressions. The definition below requires disjoint record types to simplify the
definition for our purposes, but can be altered to support overlapping record types.

Definition 24 (Predicate lossless join decomposition). $A$ lossless join decomposition of two disjoint record types $R_{1}$ and $R_{2}$ with respect to a predicate $P$ of type $R_{1} \oplus R_{2} \vdash P$ : bool, written $\boldsymbol{L J} \boldsymbol{D}\left[R_{1}, R_{2}\right](P)$, means that for all $r_{1}, s_{1} \in \operatorname{inh}\left(R_{1}\right)$ and $r_{2}, s_{2} \in \operatorname{inh}\left(R_{2}\right)$, it is the case that:

$$
\operatorname{sat}\left(P, r_{1} \otimes r_{2}\right) \wedge \operatorname{sat}\left(P, s_{1} \otimes s_{2}\right) \Longrightarrow \operatorname{sat}\left(P, r_{1} \otimes s_{2}\right)
$$

It is possible to show that this definition for predicate lossless join decomposition is consistent with the lossless join decomposition on sets. Given $R, R_{1}, R_{2}$ such that $R=R_{1} \oplus R_{2}$, our definition of lossless join decomposition suffices to show that $\operatorname{set}(P, R)$ can be expressed as the natural join of $\operatorname{set}(P, R)$ restricted to the fields of $R_{1}$, with $\operatorname{set}(P, R)$ restricted to the fields of $R_{2}$.

Lemma 40 (Predicate lossless join decomposition consistent). Suppose $R=R_{1} \oplus$ $R_{2}$ and $R \vdash P$ : bool. If $\boldsymbol{L J} \boldsymbol{D}\left[R_{1}, R_{2}\right](P)$, then $\operatorname{set}(P, R)$ forms a lossless join decomposition over $R_{1}$ and $R_{2}$.

Proof.
$R=R_{1} \uplus R_{2} \quad$ assumption
$\mathbf{L J D}\left[R_{1}, R_{2}\right](P)$ assumption

$$
\begin{align*}
& \forall r, s \in \operatorname{inh}(R) . \operatorname{sat}(P, r) \text { iff. } \operatorname{sat}\left(P, r\left[\operatorname{dom}\left(R_{1}\right)\right] \otimes s\left[\operatorname{dom}\left(R_{2}\right)\right]\right) \\
& \wedge \operatorname{sat}\left(P, r\left[\operatorname{dom}\left(R_{2}\right)\right] \otimes s\left[\operatorname{dom}\left(R_{1}\right)\right]\right) \tag{1}
\end{align*} \quad \text { def. LJD }\left[R_{1}, R_{2}\right](P)
$$

$$
\begin{array}{rlr}
\operatorname{set} & (P, R) & \\
= & \{r \mid \forall r \in \operatorname{inh}(R) \cdot \operatorname{sat}(P, r)\} & \operatorname{def.} \operatorname{set}(\cdot, \cdot) \\
= & \left\{r \mid \forall r, s \in \operatorname{inh}(R) \cdot \operatorname{sat}\left(P, r\left[\operatorname{dom}\left(R_{1}\right)\right] \otimes s\left[\operatorname{dom}\left(R_{2}\right)\right]\right)\right. & \\
& \left.\wedge \operatorname{sat}\left(P, r\left[\operatorname{dom}\left(R_{2}\right)\right] \otimes s\left[\operatorname{dom}\left(R_{1}\right)\right]\right)\right\} & \\
= & \left\{r \mid \forall r \in \operatorname{inh}(R) \cdot r\left[\operatorname{dom}\left(R_{1}\right)\right] \in\left\{s\left[\operatorname{dom}\left(R_{1}\right)\right] \mid \forall s \in \operatorname{inh}(R) \cdot \operatorname{sat}(P, s)\right\}\right. \\
& & \text { and } \left.r\left[\operatorname{dom}\left(R_{2}\right)\right] \in\left\{s\left[\operatorname{dom}\left(R_{2}\right)\right] \mid \forall s \in \operatorname{inh}(R) \cdot \operatorname{sat}(P, s)\right\}\right\} \\
= & \text { Lemma }\left[\forall r \in \operatorname{inh}(R) \cdot r\left[\operatorname{dom}\left(R_{1}\right)\right] \in \operatorname{set}(P, R)\left[R_{1}\right]\right. \text { and } & \\
& \left.r\left[R_{2}\right] \in \operatorname{set}(P, R)\left[R_{2}\right]\right\} & \operatorname{def.} \cdot[\cdot]  \tag{•}\\
= & \operatorname{set}(P, R)\left[\operatorname{dom}\left(R_{1}\right)\right] \bowtie \operatorname{set}(P, R)\left[\operatorname{dom}\left(R_{2}\right)\right] & \operatorname{def.} \cdot \bowtie .
\end{array}
$$

|  |  | $\mathrm{LJD}^{\dagger}-\mathrm{AND}$ |
| :--- | :--- | :--- |
| $\mathrm{LJD}^{\dagger}-1$ | $\mathrm{LJD}^{\dagger}-2$ | $\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](P)$ |
| $\frac{R_{1} \vdash P: \text { bool }}{\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](P)}$ | $\frac{R_{2} \vdash P: \text { bool }}{\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](P)}$ |  |
| $\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](Q)$ |  |  |
| $\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](P \wedge Q)$ |  |  |

Figure 4.2: Lossless join decomposition approximation.

Showing $\mathbf{L J D}\left[R_{1}, R_{2}\right](P)$ is NP-hard and could be undecidable, depending on the atomic formulae available in the predicates. Since a predicate that satisfies $\mathbf{L J D}\left[R_{1}, R_{2}\right](P)$ can be rewritten as a conjunction of predicates which depend only on either $R_{1}$ or $R_{2}$, we can, however, define a sound but incomplete syntactic approximation $\mathbf{L J D}{ }^{\dagger}\left[R_{1}, R_{2}\right](P)$.

The approximation in Figure 4.2 has three inference rules. If the expression $P$ can be typed under only the subset of $R_{1}$, then the predicate forms a lossless join decomposition for $R_{1}$ and $R_{2}$. Similarly, if the expression can typed under only the subset of $R_{2}$, then it also forms a lossless join decomposition. Finally if a conjunction is encountered, the conjunction forms a lossless join decomposition if each of the terms also form a lossless join decomposition.

The lossless join decomposition approximation implies a lossless join decomposition on the predicate:

Lemma 41 (Soundness of $\mathrm{LJD}^{\dagger}$ ). Given a predicate $P$ and record types $R_{1}, R_{2}$, it follows that $\boldsymbol{L} \boldsymbol{J} \boldsymbol{D}^{\dagger}\left[R_{1}, R_{2}\right](P)$ implies $\boldsymbol{L J} \boldsymbol{D}\left[R_{1}, R_{2}\right](P)$.

Proof.
$r_{1}, r_{2} \in \operatorname{inh}(R)$ assumption
$s_{1}, s_{2} \in \operatorname{inh}\left(R^{\prime}\right) \quad$ assumption
$\operatorname{sat}\left(P, r_{1} \otimes s_{1}\right) \quad$ assumption (2)
$\operatorname{sat}\left(P, r_{2} \otimes s_{2}\right)$
assumption (3)

Perform induction on $\mathbf{L J D}^{\dagger}\left[R, R^{\prime}\right](P)$

```
LJD \({ }^{\dagger}-1\)
    \(R \vdash P\) : bool
\(\overline{\mathbf{L J D}^{\dagger}\left[R, R^{\prime}\right](P)}\)
        \(\operatorname{sat}\left(P, r_{1}\right)\)
        \(\operatorname{sat}\left(P, r_{1} \otimes s_{2}\right)\)
        \(\mathbf{L J D}\left[R, R^{\prime}\right](P)\)
\(\mathrm{LJD}^{\dagger}\) - 2
    \(R^{\prime} \vdash P\) : bool
\(\mathbf{L J D}^{\dagger}\left[R, R^{\prime}\right](P)\)
        \(\operatorname{sat}\left(P, s_{2}\right)\)
        \(\operatorname{sat}\left(P, r_{1} \otimes s_{2}\right)\)
        \(\mathbf{L J D}\left[R, R^{\prime}\right](P)\)
LJD \({ }^{\dagger}\)-And
\(\mathbf{L J D}^{\dagger}\left[R, R^{\prime}\right]\left(Q_{1}\right) \quad \mathbf{L J D}^{\dagger}\left[R, R^{\prime}\right]\left(Q_{2}\right)\)
    \(\mathbf{L J D}^{\dagger}\left[R, R^{\prime}\right]\left(Q_{1} \wedge Q_{2}\right)\)
    \(\operatorname{sat}\left(Q_{1}, r_{1} \otimes s_{2}\right) \quad\) induction
    \(\operatorname{sat}\left(Q_{2}, r_{1} \otimes s_{2}\right) \quad\) induction
    \(\operatorname{sat}\left(Q_{1} \wedge Q_{2}, r_{1} \otimes s_{2}\right) \quad\) def. \(\cdot \wedge\).
    \(\mathbf{L J D}\left[R, R^{\prime}\right]\left(Q_{1} \wedge Q_{2}\right)\) def. LJD
```

Resulting from Lemma 40 and Lemma 41 , any predicate $P$ satisfying $\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](P)$ satisfies the lossless join decomposition required by the drop lens rules as defined by Bohannon et al. [12].

### 4.2.3 Default Value Check

The put semantics for the drop lens must provide a value for the dropped columns $R_{2}$. If no matching value can be found in the underlying table, the lens uses the default value $r: R_{2}$ provided by the programmer. For the lens to be well-behaved, it needs to ensure that the resulting records satisfy the underlying predicate of the lens. Therefore, in addition to showing that the predicate does not impose
any dependency between the value of the dropped field and the other fields, we must show that the default values $r$ of the dropped columns never violate the predicate $P$. Given the set representation of a predicate $\operatorname{set}(P, R)$, we must show that $r \in \operatorname{set}(P, R)\left[R_{2}\right]$.

The property $\mathbf{D V}\left[R_{1}, R_{2}\right](P, r)$ is defined, specifying that the values $r: R_{2}$ for the dropped columns never violate the predicate $P$. The idea is that there must be some record $s$ that satisfies the predicate $P$, such that $s\left[R_{2}\right]=r$. Unlike the lossless join decomposition check, $R_{1}$ and $R_{2}$ are not commutative for the default value check.

Definition 25. Given a predicate $P$ and disjoint record types $R_{1}$ and $R_{2}$ such that $\boldsymbol{L J} \boldsymbol{D}\left[R_{1}, R_{2}\right](P)$ and $r \in \operatorname{inh}\left(R_{2}\right)$, we write $\boldsymbol{D} \boldsymbol{V}\left[R_{1}, R_{2}\right](P, r)$ when $\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)$ is not empty and there exists an $s \in \operatorname{inh}\left(R_{1}\right)$ such that $\operatorname{sat}(P, r \otimes s)$.

The property $\mathbf{D V}\left[R_{1}, R_{2}\right](P, r)$ is consistent with the set semantics definition used by Bohannon et al. [12].

Lemma 42. Suppose $R=R_{1} \oplus R_{2}, r \in \operatorname{inh}\left(R_{2}\right)$, and $\boldsymbol{L J J}\left[R_{1}, R_{2}\right](P)$. Then $\boldsymbol{D} \boldsymbol{V}\left[R_{1}, R_{2}\right](P, r)$ implies $r \in \operatorname{set}(P, R)\left[R_{2}\right]$.

Proof.
$R=R_{1} \uplus R_{2} \quad$ assumption

$$
\begin{array}{rrr}
\exists s \in \operatorname{inh}\left(R_{1}\right) \cdot \operatorname{sat}(P, r \otimes s) & \text { assumption } \\
\Longrightarrow \exists s \in \operatorname{inh}(R) \cdot s\left[\operatorname{dom}\left(R_{2}\right)\right]=r \text { and } \operatorname{sat}(P, s) & \text { def. } \cdot \otimes \cdot \\
\Longrightarrow r \in\left\{s\left[\operatorname{dom}\left(R_{2}\right)\right] \mid \forall s \in \operatorname{inh}(R) \cdot \operatorname{sat}(P, r)\right\} & \text { def. } \cdot \in \cdot \\
\Longrightarrow r \in \operatorname{set}(P, R)\left[\operatorname{dom}\left(R_{2}\right)\right] & \text { def. } \cdot[\cdot]
\end{array}
$$

As with the definition of $\mathbf{L J D}\left[R_{1}, R_{2}\right](P)$, determining if $\mathbf{D V}\left[R_{1}, R_{2}\right](P, r)$ holds in the general case is NP-hard. To simplify this problem we introduce an incomplete set of inference rules to determine $\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](P, r)$ shown in Figure 4.3. which covers the same set of predicates as the $\mathbf{L} \mathbf{J D}^{\dagger}\left[R_{1}, R_{2}\right](P)$ rule. As with the lossless join decomposition, any conjunction of terms requires each term to satisfy the default value check condition. If a term does not depend on the columns that are dropped and can thus be typed under the remaining

|  | $\mathrm{DV}^{\dagger}-2$ | $\mathrm{DV}^{\dagger}-\mathrm{AND}$ |
| :--- | :---: | :---: |
| $\mathrm{DV}^{\dagger}-1$ | $R_{2} \vdash P: A$ | $\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](P, r)$ |
| $\frac{R_{1} \vdash P: A}{}$ | $\frac{\operatorname{sat}(P, r)}{\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](P, r)}$ | $\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](P, r)$ |$\quad \frac{\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](Q, r)}{\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](P \wedge Q, r)}$

Figure 4.3: Default value check rules.
columns $R_{1}$, then the term satisfies the default value check. If the term types under only the dropped columns $R_{2}$, then it must be ensured that the term evaluates to true when evaluated under the context of the default value $r$.

Lemma 43. Given a predicate $P$ such that $\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)$ is not empty and record $r$ such that $r: R$, it follows that $\boldsymbol{D} \boldsymbol{V}^{\dagger}\left[R_{1}, R_{2}\right](P, r)$ implies $\boldsymbol{D} \boldsymbol{V}\left[R_{1}, R_{2}\right](P, r)$.

Proof.
$\begin{array}{lr}r \in \operatorname{inh}\left(R^{\prime}\right) & \text { assumption } \\ \operatorname{set}\left(P, R \oplus R^{\prime}\right) \text { not empty }\end{array}$

Perform induction on $\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](\lambda x . P, r)$
DV ${ }^{\dagger}-1$
$x: R \vdash P: D$
$\overline{\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](P, r)}$
$\exists s \in \operatorname{inh}(R) . \operatorname{sat}(P, s)$
$\exists s \in \operatorname{inh}(R) . \operatorname{sat}(P, r \otimes s)$
assumption
$\mathbf{D V}\left[R_{1}, R_{2}\right](P, r)$
def. DV
$\mathrm{DV}^{\dagger}-2$
$R^{\prime} \vdash P: D \quad \operatorname{sat}(P, r)$
$\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](P, r)$
assumption

$$
\begin{array}{lr}
\exists s \in \operatorname{inh}(R) . \operatorname{sat}(P, r \otimes s) & \text { widening } \\
\mathbf{D V}\left[R_{1}, R_{2}\right](P, r) & \text { def. DV }
\end{array}
$$

$\mathrm{DV}^{\dagger}$-And
$\frac{\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right]\left(Q_{1}, r\right) \quad \mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right]\left(\lambda x . Q_{2}, r\right)}{\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right]\left(Q_{1} \wedge Q_{2}, r\right)}$

| $\exists s \in \operatorname{inh}(R) . \operatorname{sat}\left(Q_{1}, r \otimes s\right)$ | induction |
| :--- | ---: |
| $\exists s \in \operatorname{inh}(R) . \operatorname{sat}\left(Q_{2}, r \otimes s\right)$ | induction |
| $\exists s \in \operatorname{inh}(R) . \operatorname{sat}\left(Q_{1} \wedge Q_{2}, r \otimes s\right)$ | def. $\cdot \wedge$. |
| DV $\left[R_{1}, R_{2}\right](P, r)$ | def. DV |

Note that the soundness proof for $\mathbf{D} \mathbf{V}^{\dagger}\left[R_{1}, R_{2}\right](P, r)$ requires that $\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)$ is not empty. This is problematic in theory, because it requires us to show that the predicate is satisfiable. According to Bohannon et al. [12], a drop lens on a lens with predicate that is false does not typecheck. In practice however, this lens is well behaved as it returns an empty view (a view with no records, equivalent to the unit type) and only takes an empty view. The lens would therefore be trivial, but still well-behaved.

### 4.2.4 Predicate Ignores Columns

The select lens filters a view according to a given predicate. Let us assume we have a lens $l_{1}$ which is the join of the two tables albums and tracks. We might first define a lens $l_{2}$ to find popular albums for which the stock is too low, by only returning the albums where quantity < rating.

| track | date | rating | album | quantity |
| :---: | :---: | :---: | :---: | :---: |
| Lullaby | 1989 | 3 | Galore | 1 |
| Lovesong | 1989 | 5 | Galore | 1 |
| Lovesong | 1989 | 5 | Paris | 4 |
| Trust | 1992 | 4 | Wish | 4 |

We might then decide to further limit this view by defining a lens $l_{3}$ which only shows the tables with the album Galore.

| track | date | rating | album | quantity |
| :---: | :---: | :---: | :---: | :---: |
| Lullaby | 1989 | 3 | Galore | 1 |
| Lovesong | 1989 | $\boxed{5} 4$ | Galore | 1 |

The user then notices that the rating for Lovesong is not correct, and changes it from 5 to 4 . Calling put on $l_{3}$ would yield the updated view for $l_{2}$ :

| track | date | rating | album | quantity |
| :---: | :---: | :---: | :---: | :---: |
| Lullaby | 1989 | 3 | Galore | 1 |
| Lovesong | 1989 | D 4 | Galore | 1 |
| Lovesong | 1989 | D 4 | Paris | 4 |
| Trust | 1992 | 4 | Wish | 4 |

Since the rating of the track Lovesong is 4 and not lower than the quantity of the album Paris, the updated view for $l_{2}$ violates the predicate requirement quantity < rating.

To prevent such an invalid combination of lenses, the select lens needs to ensure that the underlying lens has no predicate constraints on any fields which may be changed by functional dependencies. The set of fields which can be changed by functional dependencies $F$ is outputs $(F)$. A predicate $P$ ignores the set $R$ if the result of evaluating the predicate $P$ with respect to a row in the database is not affected by changing any fields in $R$.

For lenses such as the select lens to be well-behaved, it is required to show that the predicates do not depend on the outputs of the functional dependencies. The generalisation of this is a restriction showing that a predicate does not depend on any of the columns in $R$. Bohannon et al. [12] provide the following definition for this check on sets:

Definition 26. Given $R=R_{1} \oplus R_{2}, M: R$ ignores $R_{1}$ if for any $r, s: R$ such that $r\left[R_{2}\right]=s\left[R_{2}\right]$ then $r \in M \Leftrightarrow s \in M$.

As the check should be performed on predicates, the following definition is proposed:

Definition 27. Given $R=R_{1} \oplus R_{2}$, if $R_{1} \vdash P: A$ then $P$ ignores $R_{2}$.
The following helper lemma shows that the result of predicate evaluation only depends on the columns referenced by the predicate.

Lemma 44. If $R_{1} \subseteq R$ and $R_{1} \vdash P: A$, then $P \Downarrow_{r} v$ and $P \Downarrow_{r\left[R_{1}\right]} w$ implies $v=w$.

Proof. By induction on $P$.

The definition for the ignores relation on predicates is consistent with the definition on sets:

Lemma 45 (Predicate ignores consistent). For any $R \vdash P$ : bool, if $R=R_{1} \uplus R_{2}$ and $P$ ignores $R_{1}$, then $\operatorname{set}(P, R)$ ignores $R_{1}$.

Proof.
$R=R_{1} \uplus R_{2} \quad$ assumption
$P$ ignores $R_{1}$ assumption
$\Rightarrow R_{2} \vdash P$ : bool
$r, s: R$ assumption
$r\left[R_{2}\right]=s\left[R_{2}\right]$

Perform case analysis on $r \in \operatorname{set}(P, r)$
Case $r \in \operatorname{set}(P, R)$ :

$$
\begin{array}{lr}
\operatorname{sat}(P, r) & \text { assumption, } \operatorname{defn} \operatorname{set}(\cdot, \cdot) \\
P \Downarrow_{r} \text { true } & \operatorname{defn} . \operatorname{sat}(P, r) \\
P \Downarrow_{r\left[R_{2}\right]} \text { true } & \text { Lemma } 44(1) \\
P \Downarrow_{s\left[R_{2}\right]} \text { true } & \\
P \Downarrow_{s} \text { true } & \text { Lemma } 44(1) \\
\operatorname{sat}(P, s) & \operatorname{defn} . \operatorname{sat}(P, r)
\end{array}
$$

Case $r \notin \operatorname{set}(P, R)$ :

$$
\begin{aligned}
& \operatorname{sat}(P, r) \\
& P \Downarrow_{r} \text { false } \\
& P \Downarrow_{r\left[R_{2}\right]} \text { false } \\
& P \Downarrow_{s\left[R_{2}\right]} \text { false } \\
& P \Downarrow_{s} \text { false } \\
& \operatorname{sat}(P, s)
\end{aligned}
$$

assumption, $\operatorname{defn} . \operatorname{set}(\cdot, \cdot)$
defn. $\operatorname{sat}(P, r)$
Lemma 44, 1
(2)

Lemma 44, 11
defn. $\operatorname{sat}(P, r)$

### 4.2.5 Dropping Column References

Once the predicate $P$ has passed both the default value and lossless join decomposition checks, it is necessary to remove all references to the dropped columns

Normalisation
$\llbracket P \rrbracket_{R_{1}, R_{2}}$

$$
\begin{aligned}
\llbracket P \rrbracket_{R_{1}, R_{2}} & =P & & \text { when } R_{1} \vdash P: \text { bool } \\
\llbracket P \rrbracket_{R_{1}, R_{2}} & =\text { true } & & \text { when } R_{2} \vdash P: \text { bool } \\
\llbracket P \wedge Q \rrbracket_{R_{1}, R_{2}} & =\llbracket P \rrbracket_{R_{1}, R_{2}} \wedge \llbracket Q \rrbracket_{R_{1}, R_{2}} & &
\end{aligned}
$$

Figure 4.4: Rewriting rules to remove column terms from predicate.
$R_{2}$. This is necessary so that the resulting lens type does not become ill-typed by referring to columns that do not exist in the resulting view.

As described in the introduction, removing the dropped columns can pose some issues. Luckily, using the special properties of the lossless join decomposition check makes removing references to $R_{2}$ trivial. The lossless join decomposition check ensures that the predicate is a conjunction of terms that depend either on only the dropped column or the remaining columns. This makes it possible to remove all terms that only refer to $R_{2}$.

The lens is guaranteed to produce values $r: R_{2}$ for the dropped column that will be in $P\left[R_{2}\right]$. The values $r$ will therefore satisfy any of the sub-terms $Q$ in the conjunction $P$ if $Q$ only depends on the dropped columns $R_{2}$.

Figure 4.4 shows the rewriting strategy used to remove the unwanted terms. The first entry removes any term that only references $R_{2}$ and replaces it with the expression true, which is well typed under any record type context. The second match expression leaves all terms that only reference the remaining columns $R_{1}$. The final expression recursively applies the rewriting rules to each term in the conjunction.

The rewriting rules in Figure 4.4 are not defined for arbitrary predicates. The three rules correspond to the three inference rules of $\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](P)$. This allows us to ensure that any predicate that satisfies $\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](P)$ can be rewritten. It is also possible to show that all references to $R_{2}$ are removed. We would like to show that dropping the columns as shown in Figure 4.4 is the equivalent to performing a projection on the predicate set. We start with a few helper Lemmas.

Lemma 46. If $R_{1} \vdash P$ : bool then $\operatorname{set}\left(P, R_{1}\right)=\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[R_{1}\right]$.

```
Proof.
\(\operatorname{set}\left(P, R_{1}\right)\)
    \(=\left\{r \in \operatorname{inh}\left(R_{1}\right) \mid \operatorname{sat}(P, r)\right\} \quad \operatorname{def.} \operatorname{set}\left(\cdot, R_{1}\right)\)
    \(=\left\{r\left[R_{1}\right] \in \operatorname{inh}\left(R_{1}\right) \mid \operatorname{sat}(P, r)\right\} \quad \cdot\left[R_{1}\right]\) unit
    \(=\left\{r\left[R_{1}\right] \in \operatorname{inh}\left(R_{1} \oplus R_{2}\right) \mid \operatorname{sat}(P, r)\right\} \quad\) Lemma 44
    \(=\left\{r \in \operatorname{inh}\left(R_{1} \oplus R_{2}\right) \mid \operatorname{sat}(P, r)\right\}\left[R_{1}\right] \quad\) def. []
    \(=\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[R_{1}\right] \quad \operatorname{def} . \operatorname{set}\left(\cdot, R_{1} \oplus R_{2}\right)\)
```

Lemma 47. If $M$ and $N$ are lossless join decompositions over the disjoint domains $U$ and $V$, and $M \cap N$ is not empty, then $M[U] \cap N[U]=(M \cap N)[U]$.

Proof. For any record $r: U$ we consider two cases.
If $r \in(M \cap N)[U]$, then there must exist some $s$ such that $s[U]=r$ and $s \in M \cap N$. From this we know that $s \in M$ and $s \in N$, which means that $r \in M[U]$ and $r \in N[U]$.

We then show $r \notin \operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]$ can never hold by contradiction:

$$
\begin{aligned}
& r \notin \operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right] \\
& \operatorname{set}\left(P, R_{1} \oplus R_{2}\right) \text { ignores } R_{1} \\
& \exists s \in \operatorname{set}\left(P, R_{1} \oplus R_{2}\right) \\
& r \otimes s\left[R_{2}\right] \in \operatorname{set}\left(P, R_{1} \oplus R_{2}\right) \\
& \Rightarrow r \in \operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]
\end{aligned}
$$

assumption (1)
$P$ ignores $R_{1}$ (2)
$\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)$ is not empty (3)
(2. 3); def. ignores on sets contradicts (1)

Lemma 48. If $P$ ignores $R_{1}$ such that $\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)$ is not empty, then $\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]=\operatorname{set}\left(\right.$ true,$\left.R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]$.

Proof. For any record $r: \operatorname{dom}\left(R_{1}\right)$ we consider two cases.
If $r \in \operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]$, then it is straightforward to show that $r \in \operatorname{set}\left(\right.$ true,$\left.R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]$.

We then show $r \notin \operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]$ can never hold by contradiction:

$$
\begin{aligned}
& r \notin \operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right] \\
& \operatorname{set}\left(P, R_{1} \oplus R_{2}\right) \text { ignores } R_{1} \\
& \exists s \in \operatorname{set}\left(P, R_{1} \oplus R_{2}\right) \\
& r \otimes s\left[R_{2}\right] \in \operatorname{set}\left(P, R_{1} \oplus R_{2}\right) \\
& \Rightarrow r \in \operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]
\end{aligned}
$$

assumption (1)
$P$ ignores $R_{1}$ (2) $\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)$ is not empty (3)
(2. 3); def. ignores on sets contradicts (1)

We can now show that our predicate dropping operation is equivalent to the projection operator on abstract sets. The lemma requires proof that $\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)$ is not empty. This condition is technically satisfied, because the default value condition requires such at least one entry in the predicate, meaning that it is not empty.

Lemma 49. Suppose $P$ such that $\boldsymbol{L J D}\left[R_{1}, R_{2}\right](P)$ and $R_{1} \oplus R_{2} \vdash P$ : bool such that $\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)$ is not empty. Then $\operatorname{set}\left(\llbracket P \rrbracket_{R_{1}, R_{2}}, R_{1}\right)=\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]$.

Proof.
$\mathbf{L J D}\left[R_{1}, R_{2}\right](P)$ assumption
$R \oplus\{\ell: A\} \vdash P:$ bool assumption
$\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)$ is not empty assumption (1)
perform induction on $\mathbf{L J D}\left[R_{1}, R_{2}\right](P)$
case $\mathrm{LJD}^{\dagger}-1$

$$
\frac{R_{1} \vdash P: \text { bool }}{\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](P)}
$$

$$
\operatorname{set}\left(\llbracket P \rrbracket_{R_{1}, R_{2}}, R_{1}\right)
$$

$$
=\operatorname{set}\left(P, R_{1}\right) \quad \operatorname{def} . \llbracket \cdot \rrbracket_{R_{1}, R_{2}}
$$

$$
=\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right] \quad \text { Lemma } 46
$$

case $\mathrm{LJD}^{\dagger}-2$

$$
\begin{aligned}
& \mathrm{LJD}^{\dagger}-2 \\
& \frac{R_{2} \vdash P: \text { bool }}{\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](P)}
\end{aligned}
$$

assumption
$P$ ignores $R_{1}$ def. ignores

$$
\begin{aligned}
\operatorname{set} & \left(\llbracket P \rrbracket_{R_{1}, R_{2}}, R_{1}\right) \\
& =\operatorname{set}\left(\text { true }, R_{1}\right) \\
& =\operatorname{set}\left(\text { true }, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right] \\
& =\operatorname{set}\left(P, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]
\end{aligned}
$$

$$
=\operatorname{set}\left(\operatorname{true}, R_{1}\right) \quad \operatorname{def} . \llbracket \cdot \rrbracket_{R_{1}, R_{2}}
$$

case $\mathrm{LJD}^{\dagger}$-And

| $\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right]\left(P_{1}\right) \quad \mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](Q)$ | assumption |
| :---: | :---: |
| $\mathbf{L J D}{ }^{\dagger}\left[R_{1}, R_{2}\right]\left(P_{1} \wedge Q\right)$ |  |
| $\operatorname{set}\left(P_{1} \wedge Q, R_{1} \oplus R_{2}\right)$ |  |
| $=\operatorname{set}\left(P_{1}, R_{1} \oplus R_{2}\right) \cap \operatorname{set}\left(Q, R_{1} \oplus R_{2}\right)$ | Lemma 38 |
| $\Rightarrow \operatorname{set}\left(P_{1}, R_{1} \oplus R_{2}\right)$ is not empty | (11); $\cap$ least (2) |
| $\Rightarrow \operatorname{set}\left(Q, R_{1} \oplus R_{2}\right)$ is not empty | (11); $\cap$ least (3) |
| $\operatorname{set}\left(\left[P_{1} \wedge Q \rrbracket_{R_{1}, R_{2}}, R_{1}\right)\right.$ |  |
| $=\operatorname{set}\left(\llbracket P_{1} \rrbracket_{R_{1}, R_{2}} \wedge \llbracket Q \rrbracket_{R_{1}, R_{2}}, R_{1}\right)$ | def. $\llbracket \cdot \rrbracket_{R_{1}, R_{2}}$ |
| $=\operatorname{set}\left(\llbracket P_{1} \rrbracket_{R_{1}, R_{2}}, R\right) \cap \operatorname{set}\left(\llbracket Q \rrbracket_{R_{1}, R_{2}}, R_{1}\right)$ | Lemma 38 |
| $=\operatorname{set}\left(P_{1}, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]$ |  |
| $\cap \operatorname{set}\left(Q, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]$ (2, 3) | tion hypothesis |
| $=\left(\operatorname{set}\left(P_{1}, R_{1} \oplus R_{2}\right) \cap \operatorname{set}\left(Q, R_{1} \oplus R_{2}\right)\right)\left[\operatorname{dom}\left(R_{1}\right)\right]$ | (1); Lemma 47 |
| $=\operatorname{set}\left(P_{1} \wedge Q, R_{1} \oplus R_{2}\right)\left[\operatorname{dom}\left(R_{1}\right)\right]$ | Lemma 38 |

An alternative approach to the one shown in Figure 4.4 is to substitute the default values $r$ for any column references in $R_{2}$. This is the approach described in a previous publication [51].

### 4.3 Normalisation

The basic predicate syntax introduced in this chapter is helpful for producing SQL code. However, the SQL translation requirement also forms a limitation, because
the basic predicate language does not support convenient language features such as richer data types and function abstractions. Cooper [25] shows that when the input data is flat, and the resulting output is also a flat data type, it is possible to translate away additionally used language features.

It is also convenient to declare the predicate in a slightly different format. In the basic predicate syntax, the record's fields are captured in the evaluation environment. Any $\ell$ subexpression in the predicate is a lookup of the record value in the environment. This is identical to the SQL predicate syntax where we may write the filter quantity $>2$. In this example quantity refers to the column quantity bound by the current record.

When the predicate should be expressed from within another language such as Links, the environment is already bound. This existing environment is convenient, because it allows the binding and use of other variables from within the predicate. While it would be possible to just bind the record on top of the existing environment, it makes the source of values less clear. A nicer abstraction in such situations is to require the programmer to explicitly declare the record as an input.

In the Links implementation, the programmer writes the predicate as a function of type $R \rightarrow \mathbf{b o o l}$. The predicate is then translated into a basic predicate during execution, where all variables bound by the closure are substituted for their values in the output predicate.

Given a functional programming language for predicates, we wish to show that predicates can be normalised to the basic predicate syntax. Figure 4.5 introduces additional normal forms $O$ which includes variables, constants, $\lambda$-abstractions, records whose fields are all values, record projection from a variable, conditional expressions whose subterms are all in normal form, and operations whose arguments are all in normal form. Terms in predicate normal form, ranged over by $\hat{P}$ and $\hat{Q}$, are a restriction of terms in normal forms. Terms in predicate normal form are equivalent with terms in basic predicate language in Figure 4.1, except for the representation of labels $\ell$, which are projections $x . \ell$ in predicate normal form.

Evaluation rules for this language are provided in big-step semantics. All of the evaluation rules are standard rules for a $\lambda$-calculus. Evaluation is performed with-

Syntax

$$
\begin{array}{cc}
\text { Terms } e, f, g::=x|c| \lambda x . e|(\overrightarrow{\ell=e})| x . \ell \\
& \mid \quad \text { if } e \text { then } f_{1} \text { else } f_{2} \mid \odot\{\vec{e}\} \\
\text { Types } \quad \tau::=A\left|\tau_{1} \rightarrow \tau_{2}\right|(\overrightarrow{\ell: \tau})
\end{array}
$$

Normal forms

$$
\begin{aligned}
& O::=x|c| \lambda x . O|(\overrightarrow{\ell=O})| x . \ell \\
& \mid \quad \text { if } O_{1} \text { then } O_{2} \text { else } O_{3} \mid \odot\{\vec{O}\} \\
& \hat{P}, \hat{Q}::= \\
& x . \ell|c| \odot\{\overrightarrow{\hat{P}}\} \mid \text { if } \hat{P} \text { then } \hat{Q}_{1} \text { else } \hat{Q}_{2}
\end{aligned}
$$

Normalisation

$$
e: \tau \rightsquigarrow f
$$

$$
\begin{aligned}
& (\lambda x . f) e: \tau \rightsquigarrow f[e / x] \\
& (\overrightarrow{\ell=e}) \cdot \ell: \tau \rightsquigarrow e_{\ell} \\
& \text { if true then } e \text { else } f: \tau \rightsquigarrow e \\
& \text { if false then } e \text { else } f: \tau \rightsquigarrow f \\
& \text { (if } e \text { then } f_{1} \text { else } f_{2} \text { ) } f_{3}: \tau \rightsquigarrow \text { if } e \text { then } f_{1} f_{3} \text { else } f_{2} f_{3} \\
& \text { if } e_{1} \text { then } e_{2} \text { else } e_{3}:(\overrightarrow{\ell: \tau}) \rightsquigarrow(\overrightarrow{\ell=f}) \\
& \text { with } f_{\ell}= \\
& \text { if } e_{1} \text { then } e_{2} \cdot \ell \text { else } e_{3} . \ell \\
& \text { for each } \ell \in \vec{\ell}
\end{aligned}
$$

Evaluation
$e \Downarrow v$
Values $v, w::=c|\lambda x . e|(\overrightarrow{\ell=v})$

$$
\begin{gathered}
c \Downarrow \lambda x . g \\
\frac{f \Downarrow v}{e \Downarrow v} \quad \frac{f \Downarrow v \quad g[v / x] \Downarrow w}{e f \Downarrow w} \\
\frac{e \Downarrow \text { true } \quad f \Downarrow v}{\text { if } e \text { then } f \text { else } g \Downarrow v}
\end{gathered} \frac{\left(e_{i} \Downarrow v_{i}\right)_{i}}{(\overrightarrow{\ell=e}) \Downarrow(\overrightarrow{\ell=v})} \quad \frac{e \Downarrow\left(\left(\ell_{i}=v_{i}\right)_{i \in I}\right) \quad j \in I}{e . \ell_{j} \Downarrow v_{j}}
$$

Figure 4.5: Normalisation and Evaluation
out keeping track of an environment. This differs from the basic predicate syntax, where the record is used as the evaluation environment. Function application is performed by substituting the variable within the function body.

Normalisation rules $M \rightsquigarrow N$ are a subset of the rules proposed by Cooper [25]: the first four rules are standard $\beta$-reduction rules; the fifth pushes function application inside branches of a conditional; and the sixth pushes conditional expressions inside each component of a record. Normalisation rules can be applied anywhere in a term, so we do not require congruence rules.

The rewrite system is strongly normalising.
Proposition 8 (Strong normalisation). If $\Gamma \vdash e: \tau$, then there are no infinite $\rightsquigarrow$ sequences from e.

Proof. A special case of the result shown by Cooper [25].
Static predicates refer only to constants and properties of a given record. Let $\rightsquigarrow^{*}$ be the transitive, reflexive closure of the normalisation relation. Given a variable with base record type $R$, we can show that normalisation results in a term in predicate normal form.

Proposition 9 (Normal forms). If $x: R \vdash e: \tau$ and $e \rightsquigarrow * f \nLeftarrow$, then $f$ is in normal form.

Proof. By induction on the derivation of $x: R \vdash e: \tau$. See Appendix C. 1

As a corollary, by considering only terms with type bool, we can show that predicate terms can be translated into predicate normal form.

Corollary 5 (Predicate normal form). If $x: R \vdash e$ : bool and $e \rightsquigarrow^{*} f \nsim$, then $f$ is in predicate normal form.

Consequently, for any predicate of the form $\lambda x . e$ written in our predicate language, $e$ can be normalised to predicate normal form, allowing it to be used in typechecking of lenses and for translation into SQL. The predicate normal form term $\hat{P}$ can be converted into the basic predicate language $P$ by substituting all occurrences of record projection $x . \ell$ with the label expression $\ell$.

Even though the normalisable predicate language introduced in Figure4.5 is more expressive than the basic predicate language, there are still some restrictions that apply. Recursive functions, functions with side-effects (such as printing) as well as expressions referencing any database tables are not supported.

### 4.4 Static, Dynamic and Hybrid Predicates

The predicate checks and normalisation process described in Section 4.2 and Section 4.3 require knowledge of the predicate syntax. As an example, we require that the predicate of a select lens does not refer to the outputs of the functional dependencies of a table This has implications for when and how the predicate is checked. The relational lenses implementation in Links differentiates between two types of predicates:

Static Predicates Predicates that are fully known during compile time and only rely on static information are called static predicates. An example of such a predicate is fun(a) \{ a.album == "Paris"\}, where there are no free variables. As the predicates are fully known statically, it is possible to perform all checks on the predicate during compilation, preventing any runtime errors from happening. This approach is beneficial, because relational lenses have many checks that need to be performed before the lens can be safely used, making it easy to forget some of the checks. The downside is that it is not possible for predicates to be generated during runtime or to rely on runtime values, as the predicate would be unknown during compilation.

Dynamic Predicates The alternative approach is to wait for the predicate to be generated by the program and perform the tests during execution. An example of such a dynamic predicate is fun(a) \{ a.album == albumName \} which refers to the free variable albumName. As a result of the checks not being performed during execution, any errors discovered are thrown as runtime errors. Our formal results are based on static predicates, however the same results apply for dynamic predicates (which can be treated as closed at runtime). The normalisation procedure can be applied to any dynamic predicate at runtime in order to allow the same checks to be performed dynamically.

In Links the dynamic predicates are reconstructed from the intermediate rep-
resentation of the language. Another way to achieve dynamic predicates is to allow the implicit casting of lambda expressions to expression objects, allowing the code to be analysed during runtime [10].

### 4.4.1 Hybrid Predicates

Section 4.4 introduces the concept of static and dynamic predicates. Static predicates are predicates fully known during compile time. Known predicates allow all the required checks on predicates to be made before the program is run, ensuring that the application will produce errors during execution. Dynamic predicates overcome this restriction by allowing arbitrary predicates to be constructed during runtime. All required checks on dynamic predicates are then also only performed when the program is run - at the cost of potentially introducing runtime errors during execution. Even though performing these checks results in little overhead, this approach is not ideal. The programmer may miss some cases that could result in an unsuitable predicate, making the program raise a runtime error.

Hybrid predicates combine the best of static and dynamic predicates. Hybrid predicates rely on a quotation style of language integration [21], but where the core idea is to statically track the exact underlying predicate just as with static predicates, but to then allow portions of the predicate to be erased so that only the typing information is retained. This is possible because the lossless join decomposition and default value checks can still be performed when some parts of the predicates are unknown.

Consider the Guide to Pharmacology database (GtoPdb [71]) curation example from [51]. Here we have the example table for diseases with the following record type:
(disease_id: int,name: string, description: string, type: string)

In this example we would like the user to be able to enter filters for the ID, name or type of the disease. The code should construct a predicate from this input and create a lens based on this. We assume a primitive lens handle diseases with the above declared record type. The lens has the functional dependencies disease_id $\rightarrow$ name,description,type and has no predicate restrictions.

Figure 4.6 defines the function build_diseases_lens, which produces a relational

```
1.build_diseases_lens:string }->\mathrm{ int option }->\mathrm{ string
2. }->\mathrm{ lens of (diseases, }\mp@subsup{R}{ds}{},\mp@subsup{R}{ds}{}\vdash??:\mathbf{bool},\mp@subsup{F}{ds}{}
3. build_diseases_lens = \lambda name disease_id typ.
4. let p1= if name }\not="" then erase \lceilname = \name\rfloor\rceil else erase \lceiltrue\rceil in
5. let p2= case disease_id of {}\begin{array}{ll}{\mathrm{ None }}&{=>p1;}\\{\mathrm{ Some id }}&{=>\mathrm{ erase }\lceilid=\lfloorid\rfloor\wedge@p1\rceil}\end{array}}\mathrm{ in
6. let p3= if typ\not="" then p2 else erase \lceiltyp = \typ\rfloor\wedge@p2\rceil in
7. let p=\lceil@p3\wedge description =""\rceil in
8. let l= select}p\mathrm{ from diseases in
9. drop description determined by (disease_id,"") from l
```

Figure 4.6: Hybrid predicates code example.
lens based on user input. Line 3 starts by specifying that the function takes three arguments name, disease_id and typ. The variables name and typ are both of type string and, if non-empty, should be used as a filter in the predicate. The disease_id variable has the type int option (known as Maybe in Haskell), and can either be None (also known as Nothing) or Some $i$, where $i$ is an int (equivalent to Just i in Haskell). If a disease_id is provided, it should also be used in the predicate.

Line 4 constructs the predicate $p 1$, which should filter the disease name if one is specified. When constructing the predicate, the program branches on the contents of the name variable. If name is empty, a predicate is constructed that requires the name field of each record to equal the value of the program bound name variable is constructed. Otherwise a default constant expression equal to true is returned, specifying that all records should be accepted. The syntax $\lceil\hat{e}\rceil$ is used to specify that the term ê should be a quoted expression, hence 〔true〕 constructs a quoted predicate for the constant expression true. Within any quoted predicate expression, we can escape an expression $e$ (written $\lfloor e\rfloor$ ) specifying that $e$ should be computed and the resulting value inserted as a literal into the predicate expression. In the example, the code $\lceil$ name $=\lfloor$ name $\rfloor\rceil$ constructs a predicate which checks that the record field is equal to the value computed by executing the term name, which refers to the lambda bound variable.

The type of any quoted predicate expression reflects the exact structure of the expression. Consider the expression [true] which is of type pred true. The type
of the term $\lceil$ name $=\lfloor n a m e\rfloor\rceil$ is a static predicate requiring equality between the field name and an unknown string value. The type of this expression is therefore pred $($ name $=(\cdot \vdash ?:$ string $)$ ), where $\cdot \vdash ?$ : string specifies an unknown quoted expression that does not depend on any fields and return a value of type string.

The problem with the two expressions having different types

- $\operatorname{pred}($ name $=(\cdot \vdash ?:$ string $))$ and
- pred true
is that the types cannot be unified. Since neither of these predicate types depend on any columns that will be dropped, it is safe to erase the structure of the predicate, only retaining the typing information. This is done using erase, which takes a value of type pred $S$ and yields a value of type pred $(R \vdash ?: A)$ if the typing judgement $R \vdash S: A$ holds. In this case, both predicates type to bool and would both be well-typed under the record (name: string), allowing us to erase both quoted predicates to a predicate value of type pred ((name : string) $\vdash ?$ ? bool), allowing them to be unified. As we are planning to extend the predicates with additional columns later, we actually use the following type for $p 1$ instead:

$$
\text { pred }((\text { name : string, disease_id : int,typ : string }) \vdash ?: \text { bool })
$$

For convenience we define $R_{d s}$ :

$$
\text { type } R_{d s}=(\text { name : string, disease_id: int }, \text { typ : string })
$$

Line 5 branches on the disease_id argument. If no argument is specified, then the existing predicate $p 1$ is returned. Otherwise, the predicate $p 1$ is extended by adding an additional requirement that disease_id $=i d$, where $i d$ is the actual integer value specified. The extension of an existing predicate is done by inserting a quoted predicate into a new quoted predicate expression, written @ $p 1$. Predicate insertion and the unquote operator are both similar as they both start in a quoted context and take an unquoted expression. The unquote operator expects an expression that produces a primitive value such as a value of type int, which is then inserted as an erased quoted expression. An example would be $\lceil\lfloor 5\rfloor\rceil$, which yields a predicate of type pred $(\cdot \vdash ?$ : int $)$. This is in contrast to the insertion operator, where the unquoted expression computes another quoted expression,
which is then inserted without any changes．An example of an insertion would be 「＠「577，which yields a predicate of type pred 5．The resulting type of the quoted expression 「disease＿id $=\lfloor i d\rfloor \wedge @ p 1\rceil$ thus becomes：

$$
\operatorname{pred}\left(\text { disease } \_i d=(\cdot \vdash ?: \text { int }) \wedge\left(R_{d s} \vdash ? ?: \text { bool }\right)\right)
$$

Line 6 extends $p 2$ with a filter on the typ field if a value is specified．All three predicates $p 1, p 2$ and $p 3$ all have the same predicate type pred（ $R_{d s} \vdash ?$ ：bool）． Line 7 introduces the predicate $p$ ，which has an additional restriction on the column description，requiring it to be empty．As description is the column that should be dropped from the predicate，the predicate is not erased so that the added term is retained．The additional code of the quoted expression is again just copied verbatim into the static predicate portion，yielding the following type for $p$ ：

$$
\operatorname{pred}\left(\left(R_{d s} \vdash ?: \text { bool }\right) \wedge \text { description }=" "\right)
$$

In Line 8 the select lens $l$ is constructed using the predicate $p$ ．As the under－ lying diseases lens does not have a predicate restrictions，the resulting predi－ cate restriction on $l$ is equivalent to $\left(R_{d s} \vdash ?\right.$ ：bool $) \wedge$ description $=" "$ ．The drop lens in Line 9 can be safely constructed，because both $R_{d s} \vdash ?$ ：bool and description $="$＂form a lossless join decomposition．The default value check is also satisfied because the value＂＂satisfies the predicate description $="$＂．

The resulting lens type is shown in Line 2．The lens only depends on the diseases table，and yields records of type $R_{d s}$ ．The predicate restriction on the lens would be $\left(R_{d s} \vdash ?\right.$ ：bool $) \wedge$ true，which is simplified to $R_{d s} \vdash ?$ ：bool．The functional dependencies of the lens are $F_{d s}$ ，which is defined as：

$$
F_{d s}=\text { disease } \_i d \rightarrow \text { name, type } .
$$

## 4．4．2 Language extension

We improve the basic predicate syntax from Section 4.1 by allowing predicates to partially be known statically．Instead of static or dynamic predicates，hybrid predicates are dynamic predicate values $P$ with partial predicate information tracked in the type using a static predicate $S$ ．

The following summarizes the language constructs used by the hybrid predicates language：

- Program terms $e$ are regular program terms that include elements of the simply typed lambda calculus and can switch to the quoted context $\hat{e}$ using the quotation operator 「•7.
- Quoted terms ê contain the surface syntax used to express predicates in the language. The unquote $\lfloor\cdot\rfloor$ and insert @. operators can be used to switch to the program term context. To help the reader differentiate quoted term predicate constructors, these are highlighted in gold.
- Static predicates $S$ are used to express the information known about the predicate in the program type. We highlight the static predicate constructors in red to help the reader.
- Basic predicates $P$ from Figure 4.1 are used as the underlying value representation of predicates in the language.

We start with the simply typed lambda calculus with record types and let expression syntactic sugar. Figure 4.7 extends this language with hybrid predicates.

Syntax The language terms e, $f$ can also be a quoted expression term $\lceil\hat{e}\rceil$, where $\hat{e}$ is a quoted term, or predicate erasure, written erase $\hat{e}$. Quoted terms $\hat{e}, \hat{f}$ can be any of the basic predicate terms, an unquoted term $\lfloor e\rfloor$ or a predicate insertion @ $e$.

Evaluation The evaluation of a hybrid predicate expression yields a dynamic predicate value $P$. Any basic predicate $P$ is considered a value $v$ in the language. Figure 4.8 shows the evaluation rules for the language. All of the evaluation rules either are data constructors or perform nothing and are only required for the typing rules.

The evaluation rules of a quoted term $\hat{e}$ are also shown in Figure 4.8. Each predicate term constructor $\hat{e}$ recursively evaluates all of the arguments and constructors the identical predicate term. An unquoted term $e$ is expected to yield a value $v$ of a base type and can be inserted directly into the predicate expression. A predicate insertion term @e yields a predicate value $P$, which is inserted into the basic predicate.

Typing The language extension introduces the statically known predicate information $S, T$. Static predicate information contains a constructor for each basic

Syntax
Terms $\quad e, f::=\ldots \mid$ erase $e \mid\lceil\hat{e}\rceil$
Quoted Terms $\hat{e}, \hat{f}::=c|\ell|$ if $\hat{e}$ then $\hat{f}_{1}$ else $\hat{f}_{2}|\odot \vec{e}|\lfloor e\rfloor \mid @ e$
Values
$v::=\cdots \mid P$
Types
$\tau::=\cdots|A| \operatorname{pred} S$

$$
S, T::=R \vdash ?: A|c| \ell|\odot \vec{S}| \text { if } S \text { then } T_{1} \text { else } S_{2}
$$

Static predicate typing rules
T-S-IF

$$
\begin{array}{lcc}
\begin{array}{c}
\text { T-S-ConST } \\
c: \tau_{c}
\end{array} & \begin{array}{c}
\text { T-S-LABEL } \\
R \vdash c: \tau_{c}
\end{array} & \frac{(\ell: A) \in R}{R \vdash \ell: A}
\end{array} \frac{R \vdash T_{1}: A}{R \vdash \text { if } S \text { then } T_{1} \text { else } T_{2}: A} \begin{aligned}
& R \vdash T_{2}: A \\
& \hline
\end{aligned}
$$

Quoted typing rules
$\Gamma \vdash \hat{e}: \tau$

$$
\begin{aligned}
& \text { T-Q-Unquote } \\
& \frac{\Gamma \vdash e: A}{\Gamma \vdash\lfloor e\rfloor: \operatorname{pred}(\cdot \vdash \square ?: A)} \\
& \text { T-Q-IF } \\
& \text { T-Q-Label } \\
& \Gamma \vdash \ell: \operatorname{pred} \ell \\
& \text { T-Q-Insert } \\
& \frac{\Gamma \vdash e: \operatorname{pred} S}{\Gamma \vdash @ e: \operatorname{pred} S} \\
& \text { T-Q-Const } \\
& \Gamma \vdash c: \operatorname{pred} c \\
& \Gamma \vdash \hat{e}: \operatorname{pred} S \\
& \Gamma \vdash \hat{f}_{1}: \operatorname{pred} T_{1} \quad \Gamma \vdash \hat{f}_{2}: \operatorname{pred} T_{2} \\
& \overline{\Gamma \vdash \text { if } \hat{e} \text { then } \hat{f}_{1} \text { else } \hat{f}_{2}: \text { pred if } S \text { then } T_{1} \text { else } T_{2}}
\end{aligned}
$$

$$
\begin{aligned}
& \text { T-Q-OP } \\
& \frac{\odot: A_{1} \times \ldots \times A_{n} \rightarrow A \quad\left(\Gamma \vdash \hat{e}_{i}: \operatorname{pred} S_{i}\right)_{i \in 1 . . n}}{\Gamma \vdash \odot \vec{e}: \operatorname{pred} \odot \vec{S}}
\end{aligned}
$$

Typing rules

| T-H-Erase <br> $R \vdash S: A \quad \Gamma \vdash e: \operatorname{pred} S$ | T-H-Quote <br> $\Gamma \vdash$ erase $e: \operatorname{pred}(R \vdash \boxed{?}: A)$ | $\frac{\Gamma \vdash \hat{e}: \operatorname{pred} S}{\Gamma \vdash\lceil\hat{e}\rceil: \operatorname{pred} S}$ |
| :---: | :---: | :---: |

Figure 4.7: Language extensions for hybrid predicates.


Figure 4.8: Hybrid predicate translation.
predicate constructor, and these are highlighted in red to differentiate from other identically named constructors. Instead of the erase constructor, static predicate information can include an unknown predicate $R \vdash ?: A$, which specifies that the unknown predicate types to $A$ under the row $R$.

The program types $\tau$ include all basic types $\tau$ as well as an additional constructor pred $S$. This type is used to designate a hybrid predicate with static predicate information $S$.

All program terms that correspond to basic predicate constructors type to a hybrid predicate term pred $S$, where $S$ is identical to the same basic predicate constructor. The typing rule T-H-Erase for a term erase $e$ requires $e$ to type to a hybrid predicate pred $S$ such that the typing judgement $R \vdash S: A$ is satisfied, and yields the type pred $(R \vdash ?: A)$. The T-H-Quote typing rule yields the same type as the provided quoted term.

The T-Q-Unquote term expects the unquoted term $e$ to compute a value of primitive type $A$ and yields a predicate of type pred $(\cdot \vdash ?: A)$, specifying that it is an erased expression under the empty record yielding a value of type $A$. The T-Q-Insert rule expects an unquoted term $e$ to yield a predicate of type pred $S$ and the resulting type of an inserted predicate is also pred $S$.

| C-Const |  | C-Label | C-IF |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  | $P:: S$ | $Q_{1}:: T_{1}$ | $Q_{2}:: T_{2}$ |
| $c: \because c$ |  |  | $\ell:: \ell$ | if $P$ | en $Q_{1}$ e | e $Q_{2}::$ if | then $T_{1}$ else $T_{2}$ |
| C-Op |  |  |  |  |  | Rased |
| $\odot:: A_{1} \times \ldots \times A_{n} \rightarrow A$ |  |  |  | $\left.S_{i}\right)_{i \in 1 . . n}$ |  | $R \vdash P: A$ |
| $\odot\{\vec{P}\}:: \odot \vec{S}$ |  |  |  |  |  | $(R \vdash \square ? A)$ |

Figure 4.9: Predicate consistency relation.

Note that the typing rules themselves do not immediately require the predicate itself to be well-typed. An ill-formed predicate such as $5=$ true could be constructed as long as it is never used. Instead, the predicate is only checked when used as an argument to erase • or as an argument to a lens.

Any closed and well-typed terms yielding a predicate type will produce basic predicate expressions when evaluated. It is sufficient to show this for closed predicate terms, as we assume a big-step evaluation strategy where terms are substituted and evaluation always happens without any free variables.

Lemma 50 (Predicate preservation). If $\cdot \vdash e:$ pred $S$ and $e \Downarrow v$, then $v$ is a basic predicate.

Proof. By induction on $\cdot \vdash e:$ pred $S$.
Evaluation is also defined on static predicates $S$, using the same rules as defined in Figure 4.1. Note that there is no evaluation rule for unknown predicates. Instead, static predicate evaluation is only defined on predicates without any unknowns.

Consistency For any hybrid predicate term $\cdot \vdash e:$ pred $S$, the static predicate information $S$ acts as a restriction on the predicate $P$ that the term $e$ computes. To simplify the proofs needed, Figure 4.9 introduces a relation $P:: S$ which specifies that the predicate $P$ is consistent with the static predicate $S$. If $S$ is any of the basic predicate constructors, then $P$ will be that same constructor. If $S$ is an erased predicate $R \vdash ?$ : $A$, then $P$ has the type $A$ under the type context $R$.

If the static predicate $S$ types to $A$ under the context $R$, then any consistent predicate $P:: S$ will also type to $A$ under the context $R$.

Lemma 51 (Consistent Typing). If $R \vdash S: A$ and $P:: S$ then $R \vdash P: A$.

Proof.
$R \vdash S: A \quad$ assumption (1)
$P$ :: $S$
assumption (2)

Perform induction on (1):
case: T-S-Erased

$$
\frac{R_{1} \subseteq R}{R \vdash\left(R_{1} \vdash ?: A\right): A}
$$

by inversion on (2):

$$
\frac{R_{1} \vdash P: A}{P::\left(R_{1} \vdash ? ?: A\right)}
$$

$R \vdash P: A$
Lemma 35
For remaining cases see Appendix C. 2

It is also necessary to show that any predicate term $e$ that constructs a predicate of type pred $S$ will evaluate to a consistent predicate $P:: S$. The proof for this property makes use of Lemma 51.

Lemma 52 (Static predicate consistent). For any closed term e (or term ê) such that $\cdot \vdash e:$ pred $S$ and $R \vdash S: A$, if $e \Downarrow P$ then $P:: S$.

Proof.
$e: \operatorname{pred} S \quad$ assumption
$R \vdash S: A \quad$ assumption (1)
$e \Downarrow P$
assumption (2)

Perform induction on $e: \operatorname{pred} S$ :
case: T-H-Erase
$\frac{R_{1} \vdash T: A \quad f: \operatorname{pred} T}{\text { erase } f: \operatorname{pred}\left(R_{1} \vdash ?: A\right)}$

| $\frac{f \Downarrow P}{\text { erase } f \Downarrow P}$ | E-H-Erase $\sqrt{2}]$ |
| :--- | ---: |
| $P:: T$ | IH |
| $R_{1} \vdash P: A$ | Lemma 51 |
| $P::\left(R_{1} \vdash ?: A\right)$ | C-Erased |

For remaining cases see Appendix C.2

For any consistent predicate $P:: S$, if the static predicate evaluates to some value $v$ given a row $r$, then the dynamic predicate equivalent will evaluate to the same value $v$ when given the same row $r$. This property is required to show correctness of the default value checks.

Lemma 53. Given $P:: S$ such that $R \vdash S: A$ as well as $r: R$. If $S \Downarrow_{r} v$, then $P \Downarrow_{r} v$.

Proof. As $S \Downarrow_{r} v$ is defined, $S$ does not contain any erased predicates. Given the static predicate $S$ does not contain any erased predicates, it must be identical to the predicate $P$ and thus must evaluate to the same value $v$, because static predicates and basic predicates share the same evaluation rules.

### 4.4.3 Checking Hybrid Predicates

The reason for introducing hybrid predicates is the desire to perform the lossless join decomposition and default value checks described in Section 4.2 on partially known predicates. It is necessary to define these checks on static predicates, and to then show that they imply these properties on the recovered basic predicates.

Lossless join decomposition Figure 4.10 shows how to adjust the definition for the lossless join decomposition from Figure 4.2. The rules are unchanged apart from the use of static predicates. It is important to note that the LJD ${ }^{\dagger}-1$ and LJD ${ }^{\dagger}-2$ rules are well-defined on unknown predicates, as such predicates can still be typed and this is the only requirement to check if a predicate is a lossless join decomposition.

Using the lossless join decomposition on static predicate information, we can show that any consistent basic predicate will also form a lossless join decomposition.
$\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](S)$
$\mathrm{LJD}^{\dagger}$-S-1
$\mathrm{LJD}^{\dagger}$-S-2
$\frac{R_{1} \vdash S: A}{\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](S)}$
$\frac{R_{2} \vdash S: A}{\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](S)}$

LJD ${ }^{\dagger}$-S-AND
$\frac{\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](S) \quad \mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](T)}{\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](S \wedge T)}$

Figure 4.10: Lossless join decomposition inference rules for static predicates.

Lemma 54 (Hybrid Lossless Join Decomposition Consistent). If $P$ :: $S$ such that $\boldsymbol{L} \boldsymbol{J} \boldsymbol{D}^{\dagger}\left[R_{1}, R_{2}\right](S)$, then $\boldsymbol{L} \boldsymbol{J} \boldsymbol{D}^{\dagger}\left[R_{1}, R_{2}\right](P)$.

Proof.
$P:: S$
assumption (1)
$\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right](S)$
assumption

Perform induction on $\mathbf{L J D}{ }^{\dagger}\left[R_{1}, R_{2}\right](S)$

LJD ${ }^{\dagger}-\mathrm{S}-1$ and $\mathrm{LJD}^{\dagger}$-S-2 follow from Lemma 51
case: $\mathrm{LJD}^{\dagger}$-S-AND

$$
\frac{\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right]\left(S_{1}\right) \quad \mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right]\left(S_{2}\right)}{\mathbf{L J D}^{\dagger}\left[R_{1}, R_{2}\right]\left(S_{1} \wedge S_{2}\right)}
$$

assumption
by inversion on $P:: S_{1} \wedge S_{2}$

$$
\frac{\wedge: \text { bool } \times \text { bool } \rightarrow \text { bool } \quad P_{1}:: S_{1} \quad P_{2}:: S_{2}}{P_{1} \wedge P_{2}: S_{1} \wedge S_{2}}
$$

C-Op (1)
$\mathbf{L J D}{ }^{\dagger}\left[R_{1}, R_{2}\right]\left(P_{1}\right)$
$\mathbf{L} \mathbf{J D}^{\dagger}\left[R_{1}, R_{2}\right]\left(P_{2}\right)$ IH
$\mathbf{L} \mathbf{J D}^{\dagger}\left[R_{1}, R_{2}\right]\left(P_{1} \wedge P_{2}\right)$
$L^{\prime} D^{\dagger}$-And

Given any term $\cdot \vdash e: S$ that evaluates to a consistent predicate $P$, if the lossless join decomposition property applies to the static predicate $S$, then the predicate $P$ must also be a lossless join decomposition.

Lemma 55 (Hybrid Lossless Join Decomposition Consistent). For any e: pred $S$ such that $\boldsymbol{L} \boldsymbol{J} \boldsymbol{D}^{\dagger}\left[R_{1}, R_{2}\right](S)$ and $R \vdash S: A$, if $e \Downarrow P$ then $\boldsymbol{L} \boldsymbol{J}^{\dagger}\left[R_{1}, R_{2}\right](P)$.

Proof. Lemma 52 allows us to show predicate consistency $P:: S$. Using this property, Lemma 54 can be used to show that the lossless join decomposition property applies to the derived predicate.

Default value check Figure 4.11 adapts the default value check to static predicates. The rules are identical to those on regular predicates in Figure 4.3. The $\mathrm{DV}^{\dagger}-1$ rule is well-defined on predicates that have erased sub-terms while $\mathrm{DV}^{\dagger}-2$ is not, because predicate evaluation is not defined on dynamic predicates ${ }^{1}$

## $\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](S, r)$

$$
\begin{array}{cc}
\mathrm{DV}^{\dagger}-\mathrm{S}-1 & \mathrm{DV}^{\dagger}-\mathrm{S}-2 \\
R_{2} \vdash S \text { : bool } \\
R_{1} \vdash S: A & \frac{S \Downarrow_{r} \text { true }}{} \\
\hline \mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](S, r) & \mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](S, r) \\
\frac{\mathrm{DV}^{\dagger}-\mathrm{S}-\mathrm{AND}}{} & \\
\frac{\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](S, r)}{\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right]((S \wedge T), r)} & \mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](T, r)
\end{array}
$$

Figure 4.11: Default value check for static predicates.

Just as with the lossless join decomposition check, we can show that the default value check on static predicate information implies the default value check on the underlying predicates.

Lemma 56. If for any $P:: S$ such that $\boldsymbol{D} \boldsymbol{V}^{\dagger}\left[R_{1}, R_{2}\right](S, r)$, then $\boldsymbol{D} \boldsymbol{V}^{\dagger}\left[R_{1}, R_{2}\right](P, r)$.

Proof.
$P:: S$
$\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](S, r)$

Perform induction on $\mathbf{D} V^{\dagger}\left[R_{1}, R_{2}\right](S, r)$.

[^1]$\mathrm{DV}^{\dagger}$-S-1 follows from Lemma 51 .
case: $\mathrm{DV}^{\dagger}-\mathrm{S}-2$
\[

$$
\begin{array}{lr}
\quad R_{2} \vdash S \text { : bool } & \\
\frac{S \Downarrow_{r} \text { true }}{} \begin{array}{ll}
\mathrm{DV}^{\dagger}\left[R_{1}, R_{2}\right](S, r) & \text { assumption } \\
R_{2} \vdash P: \text { bool } & \text { Lemma } 51 \\
P \Downarrow_{r} \text { true } & \text { Lemma } 53 \\
\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](P, r) & \mathrm{DV}^{\dagger}-2 \\
\text { ase: } \mathrm{DV}^{\dagger} \text {-S-AND } &
\end{array} \$ . \begin{array}{l} 
\\
\hline
\end{array} &
\end{array}
$$
\]

$$
\frac{\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right]\left(S_{1}, r\right) \quad \mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right]\left(S_{2}, r\right)}{\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right]\left(\left(S_{1} \wedge S_{2}\right), r\right)}
$$

by inversion on $P::\left(S_{1} \wedge S_{2}\right)$
$\frac{\wedge: \text { bool } \times \text { bool } \rightarrow \text { bool } \quad P_{1}:: S_{1} \quad P_{2}:: S_{2}}{P_{1} \wedge P_{2}: S_{1} \wedge S_{2}}$
C-Op (1)
$\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right]\left(r, P_{1}\right)$
IH
$\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right]\left(r, P_{2}\right)$
IH
$\mathrm{DV}^{\dagger}\left[R_{1}, R_{2}\right]\left(r, P_{1} \wedge P_{2}\right) \quad \mathrm{DV}^{\dagger}$-And

Theorem 10. For any e: pred $S$ such that $\boldsymbol{D} \boldsymbol{V}^{\dagger}\left[R_{1}, R_{2}\right](S, r)$ and $R \vdash S: A$, if $e \Downarrow P$ then $\boldsymbol{D} \boldsymbol{V}^{\dagger}\left[R_{1}, R_{2}\right](P, r)$.

Proof. Lemma 52 allows us to show predicate consistency $P:: S$. Using this property, Lemma 56 can be used to show that the lossless join decomposition property applies to the derived predicate.

### 4.5 Summary

This chapter focuses on relational lens predicates. Predicates are expressions yielding a binary outcome for any given record. When defining a select lens, the programmer is required to specify a predicate used to filter out records from the resulting view. Predicates are also used in the lens refinement type, specifying which records may appear in the view. The lens typing rules depend on the predicates of the input lenses to determine if the lens can be safely constructed.

Bohannon et al. [12] define relational lenses using abstract sets. This representation of sets is convenient for defining the checks required by the relational lenses typing rules, but is insufficient for an implementation.

In this chapter, we first introduce a basic predicate syntax and show how it relates to an equivalent abstract set. Basic predicates are trivially translatable into an SQL where clause. We show how the lossless join decomposition, default value and ignores checks can be performed on the basic predicate syntax. We provide proof that the checks on basic predicates imply the corresponding property on the equivalent set predicates. The predicate syntax is then extended to support additional language features such as function abstractions and tuple types, by adapting work from Cooper [25]. We show that in the use cases required by predicates, all of these additional language features can be desugared completely.

Lastly, this chapter discusses the trade-offs between performing the introduced predicate checks during compilation and the increased flexibility of deferring the checks to run-time. Based on these observations we introduce hybrid predicates, combining the advantages from both approaches. Hybrid predicates statically track the structure of the underlying predicate constructed at runtime, but allow portions of the predicate to be erased. The hybrid predicate retains the type information of the erased predicate, and allows the predicate checks to be performed if sufficient static information is available. We show that if a hybrid predicate satisfies the required checks, then the underlying dynamic predicate used by the relational lens also must satisfy the equivalent checks on basic predicates.

To keep this chapter simple, it restricts its focuses to the predicates of relational lenses. Chapter 5 applies the newly defined predicate checks to provide a complete language integration, showing that the work on predicates in this chapter is sufficient to provide a concrete predicates implementation. Chapter 6 demonstrates how hybrid predicates are implemented in the Haskell library.

## Chapter 5

## Language Integrated Relational Lenses

Relational lenses as defined by Bohannon et al. [12] construct a lens mapping from one schema to another. Consider the introductory example from Section 1.1. The application takes the two tables albums and tracks and constructs a lens that joins, filters and projects them. In the sequential style of Bohannon et al.'s relational lenses, the lens is defined as follows:

```
join_dl albums with tracks as joined;
drop date determined by (track,2018) from joined as dropped;
select \(_{\text {quantity }>2}\) from dropped as filtered
```

In each declaration such as join_dl albums with tracks as joined, the lens expression consumes views in the schema (in this case albums and tracks), and replaces them with the newly defined view (joined). The lenses are composed using simple lens composition as defined in Section 2.2. The complete example produces a lens of type $\Sigma \uplus\{$ albums; tracks $\} \Leftrightarrow \Sigma \uplus\{$ filtered $\}$ for any database schema $\Sigma$ that does not contain albums, tracks, filtered, dropped or joined.

There are a few disadvantages of sequential relational lenses in the context of a functional programming language. Sequential lenses introduce an additional namespace for relation names, even though our language already has variable binding for referring to other values. They also require us to provide names for each intermediate lens, even though these names are internal and unobservable.

Finally, sequential lenses map multiple views to multiple views. The use case required by our programming language always assumes a fixed database schema and we are generally only interested in a single output view, giving preference to a simpler representation. If we are interested in the input view, function abstractions that take a lens and produce a lens can be used.

These sequential lenses reference views in the schema by name. A more idiomatic way of constructing relational lenses for our purposes would be to use sub-expressions in a functional language, such as Links or Haskell, yielding a lens to the corresponding table. The same lens could be constructed as follows:
let joined $=$ join_dl albums with tracks in
let dropped $=$ drop date determined by (track,2018) from joined in
select $_{\text {quantity }}{ }^{2}$ from dropped

In the example, the views are let bound to regular programming variables rather than introducing a special namespace for views in the schema. Just as one would expect with functional expressions, it also becomes possible to avoid naming lenses at all. The definition of dropped and joined can be inlined into the select lens expression:

```
select }\mp@subsup{\mathrm{ quantity>2 }}{}{\mathrm{ from}
    (drop date determined by (track,2018) from
        (join__dl albums with tracks))
```

In this section, we adapt the rules for constructing relational lenses by Bohannon et al. [12] to the setting of a functional programming language. Our adaptation is required to translate compositional to sequential relational lenses in the presence of function abstractions. We show that in our formulation, well-typed lens expressions can be evaluated to lens values that can be translated to well-behaved sequential lenses during runtime. Our language has total get and put operations, preserves types during execution and also satisfies round-tripping guarantees provided by lenses.

This Section is outlined as follows:

1. Section 5.1 presents the sequential relational lens typing rules as defined by Bohannon et al. [12].

| Set predicates | $M, N$ |  |
| :--- | :--- | :--- |
| Schemas | $\Sigma, \Delta$ |  |
| Sequential lenses | $I, J$ | $=\mathbf{i d}\|I ; J\|$ select $_{M} S$ as $T \mid$ rename $_{A / B} S$ as $T$ |
|  |  | $\mid$ join_dl $S_{1} S_{2}$ as $T \mid \operatorname{join}_{M, N} S_{1} S_{2}$ as $T$ |

Figure 5.1: Syntax of sequential relational lenses.
2. Section 5.2 introduces our language-integrated lens typing rules.
3. Section 5.3 briefly introduces the evaluation rules used by our language.
4. Section 5.4 presents a correct translation from a lens constructed in our language into a sequential lens.
5. Section 5.5 shows that evaluation in our language preserves typing and that round-tripping laws are satisfied.
6. Section 5.6 describes how relational lenses are integrated into the Links programming language.
7. Section 5.7 summarizes this chapter.

### 5.1 Sequential Lenses

Before introducing our language-integrated lenses, we first recapitulate the lens typing rules as defined by Bohannon et al. [12]. We let $S, T$ range over relation names; $\Sigma, \Delta$ range over schemas (i.e., sets of relation names); and $I, J$ range over sequential-style lenses. Sequential lenses represent predicates as sets, hence we use $M, N$ to refer to them. The sort of a relation $S$, written $\operatorname{sort}(S)=(U, M, F)$, is a 3-tuple of the set of fields $U$ in $S$; a set predicate $M$, and the set of functional dependencies $F$. If $\operatorname{sort}(S)=(U, M, F)$, then $\operatorname{dom}(S)=U$.

The constructors for sequential-style lenses are shown in Figure 5.1. These constructors are like the lens constructors shown in Section 2.3 except for the following changes:

1. Each lens explicitly mentions which relations are removed from the schema and added to the schema.
2. Product lenses are not required, because lenses always map from a single schema to a single schema.

The id lens defines the identity lens, mapping a schema to itself, and $I ; J$ composes lenses $I$ and $J$. The select ${ }_{M} S$ as $T$ lens filters relation $S$ using predicate set $M$, naming the resulting relation $T$. The join_dl $S_{1} S_{2}$ as $T$ lens joins relations $S_{1}$ and $S_{2}$ using the delete-left strategy, naming the resulting relation $T$. The join template lens join ${ }_{M, N} S_{1} S_{2}$ as $T$ supports arbitrary deletion semantics by specifying the predicates $M$ and $N$. The rename lens rename ${ }_{A / B} S$ as $T$ renames the column $A$ in the source view $S$ to $B$ in the resulting view $T$.

Finally, drop $\ell$ determined by $(U, v)$ from $S$ as $T$ drops attribute $\ell$ determined by attributes $U$ with default value $v$ from relation $S$ as $T$ in the output.

Typing Relational lenses are transformations $\Sigma \Leftrightarrow \Delta$, which consume some view from $\Sigma$ and introduce a new view in the output schema $\Delta$. Any schema can only have a single mapping for any table name. In a similar manner, the put direction of a lens can only output a single view for any table name. For lenses that combine multiple tables, such as the join lens, it is necessary to take caution that any view is used at most once. Otherwise, the put operation would also produce two variants of the same view which cannot be unified.

Figure 5.2 shows the typing rules. The sequential lens typing judgement has the shape $I \in \Sigma \Leftrightarrow \Delta$, meaning that $I$ is a lens transforming the source schema $\Sigma$ into the view schema $\Delta$. The use of schemas makes the rules slightly more restrictive than those defined in Section 2. Each typing rule must ensure the schemas are disjoint from the relation names handled by the lens.

In the case of the select lens, given a predicate set $M$, the typing rule enforces the invariant that the source relation $S$ has sort $(U, N, F)$; that the functional dependencies $F$ are in tree form; that $N$ ignores the outputs of $F$; and assigns the view $T$ the sort $(U, M \cap N, F)$.

The join lens typing rule requires both input tables to have functional dependencies in tree form, and ensures that their predicates ignore the outputs of the corresponding functional dependencies. The functional dependencies require the domain of the right table to be transitively covered by the join columns. Any record should either be in the deletion set $P_{d}$ or $Q_{d}$.

Typing rules

$$
\begin{aligned}
& \operatorname{sort}(S)=(U, N, F) \quad \operatorname{sort}(T)=(U, M \cap N, F) \\
& \begin{array}{c}
F \text { is in tree form } \quad N \text { ignores outputs }(F) \\
\text { select }_{M} S \text { as } T \in \Sigma \uplus\{S\} \Leftrightarrow \Sigma \uplus\{T\}
\end{array} \text { T-SELECT-RL } \\
& \operatorname{sort}\left(S_{1}\right)=(U, M, F) \quad \operatorname{sort}\left(S_{2}\right)=(V, N, G) \\
& S_{1} \neq S_{2} \quad \operatorname{sort}(T)=(U V, M \bowtie N, F \cup G) \quad G \vDash U \cap V \rightarrow V \\
& F \text { is in tree form } \quad G \text { is in tree form } \\
& M \text { ignores outputs }(F) \quad N \text { ignores outputs }(G) \\
& \text { join_dl } S_{1} S_{2} \text { as } T \in \Sigma \uplus\left\{S_{1}, S_{2}\right\} \Leftrightarrow \Sigma \uplus\{T\} \\
& \operatorname{sort}\left(S_{1}\right)=(U, M, F) \quad \operatorname{sort}\left(S_{2}\right)=(V, N, G) \\
& S_{1} \neq S_{2} \quad \operatorname{sort}(T)=(U V, M \bowtie N, F \cup G) \\
& G \vDash U \cap V \rightarrow V \quad M_{d} \cup N_{d}=\top_{U V} \\
& F \text { is in tree form } \quad G \text { is in tree form } \\
& \frac{M \text { ignores outputs }(F) \quad N \text { ignores outputs }(G)}{\operatorname{join}_{M_{d}, N_{d}} S_{1} S_{2} \text { as } T \in \Sigma \uplus\left\{S_{1}, S_{2}\right\} \Leftrightarrow \Sigma \uplus\{T\}} \text { T-Join-RL } \\
& \operatorname{sort}(S)=(U, M, F) \quad \operatorname{sort}(T)=(U-A, M[U-A], G) \\
& A \in U \quad F \equiv G \cup X \rightarrow A \\
& M=M[U-A] \bowtie M[A] \quad\{A=a\} \in M[A] \\
& \text { drop } A \text { determined by }(X, a) \text { from } S \text { as } T \in \Sigma \uplus\{S\} \Leftrightarrow \Sigma \uplus\{T\} \\
& U \cap\{A, B\}=\varnothing \\
& \frac{\operatorname{sort}(S)=(U A, M, F) \quad \operatorname{sort}(T)=\left(U B, \rho_{A / B}(M), F[A / B]\right)}{\text { rename }_{A / B} S \text { as } T \in \Sigma \uplus\{S\} \Leftrightarrow \Sigma \uplus\{T\}} \text { T-RENAME-RL } \\
& \frac{I \in \Sigma \Leftrightarrow \Sigma^{\prime} \quad J \in \Sigma^{\prime} \Leftrightarrow \Delta}{I ; J \in \Sigma \Leftrightarrow \Delta} \text { T-Compose-RL } \quad \overline{i d \in \Sigma \Leftrightarrow \Sigma} \text { T-Id-RL }
\end{aligned}
$$

Figure 5.2: Sequential typing rules for relational lenses.

The drop lens removes references to the column from the record type, the functional dependencies as well as the predicate. The lens requires the predicate to satisfy the lossless join decomposition and default value checks. It also ensures the functional dependencies require the dropped column to be defined by the key provided by the user.

The rename lens ensures that the new column name is not already used in the record type. It renames all references from the old column name to the new column name in the record type, functional dependencies and predicate.

### 5.2 Lens Types

We now introduce the rules we use to type check relational lenses. These rules are adapted from the rules as defined by Bohannon et al. [12] to support nested composition and to make use of our concrete predicate syntax. We define a simply typed lambda calculus, along with records and sets and extend it with relational lenses.

Figure 5.3 shows the types and terms for our language. A type $\tau$ can be a constant type $A$, a function type $\tau_{1} \rightarrow \tau_{2}$, a set of another type $\{\tau\}$, a record type $(\overrightarrow{\ell: \tau})$ or a lens type. We sometimes use $R$ as a placeholder for a record type. The type of lenses, lens of $(\Sigma, R, P, F)$, consists of four components: the set of underlying tables $\Sigma$; the base record type $R$; a restriction predicate $P$; and a set of functional dependencies $F$. Predicates $P, Q$ are basic predicates as defined in Chapter 4. The base record type describes the type of rows which can be retrieved or committed to the view, and the restriction predicate describes the subset of records on which the lens operates.

The languages terms ( $e$ and $f$ ) can either be a constant value $c$, a variable $x$, a function abstraction $\lambda x$. e, function application $e f$, constant sets $\{\vec{e}\}$, or constant records $(\overrightarrow{\ell=e})$, any of the lens constructors, a get operation or a put operation. Both constants and lambda expressions, as well as lens constructors that only refer to other values, are values. In practice, a language would likely want to include additional terms for record projection and additional set operations. As we are mainly interested in the translation of lenses, we exclude them to keep our language simple.


Figure 5.3: Syntax of types and terms for tables and lenses

$$
\begin{array}{cc}
\frac{c \text { is of type } A}{\Gamma \vdash c: A} \mathrm{~T}-\mathrm{ConsT} & \frac{(x=\tau) \in \Gamma}{\Gamma \vdash x: \tau} \mathrm{T}-\mathrm{VAR} \\
\frac{\Gamma \vdash e: \tau_{1} \rightarrow \tau_{2} \quad \Gamma \vdash f: \tau_{1}}{\Gamma \vdash e f: \tau_{2}} \mathrm{~T}-\mathrm{APP} & \frac{\Gamma \leftarrow\left(x=\tau_{1}\right) \vdash e: \tau_{2}}{\Gamma \vdash \lambda x . e: \tau_{1} \rightarrow \tau_{2}} \mathrm{~T}-\mathrm{ABS} \\
\frac{\left(\Gamma \vdash e_{i}: \tau_{i}\right)_{i \in 1 \ldots n}}{\Gamma \vdash(\overrightarrow{\ell=e}):(\overrightarrow{\ell: \tau})} \mathrm{T}-\mathrm{RECORD} & \frac{\left(\Gamma \vdash e_{i}: \tau\right)_{i \in 1 \ldots n}}{\Gamma \vdash\{\vec{e}\}:\{\tau\}} \mathrm{T}-\mathrm{Const}-\mathrm{SET}
\end{array}
$$

Figure 5.4: Typing rules.

We now present the typing judgements $\Gamma \vdash e: \tau$ for our relational lenses language. The typing judgement indicates that the term $e$ types to $\tau$ under the context $\Gamma$. We assume that the typing context $\Gamma$ is a record from labels to types. An implicit and global database schema $\varphi$ is assumed in all the typing rules. The database schema maps relation names $S$ to relation types $\operatorname{Rel}(R, P, F)$. We assume that any database context $\varphi$ is an instance of the schema $S$, meaning that for any $S=\operatorname{Rel}(R, P, F)$ in $\Phi$ the database context $\varphi$ must include a relation $S=M$ such that $M: \operatorname{Rel}(R, P, F)$. Figure 5.4 presents the typing rules for constants, variables function abstractions, sets and records.

In the remainder of the section, we describe each lens combinator and its typing rule in turn.

### 5.2.1 Lens Primitive

The rule T-Lens is used to create an identity lens referring to an actual table on the database. The programmer specifies the name of the table $S$ which is recorded in the type of the lens. A lens primitive is assigned the default predicate constraint true. All columns referred to by a set of functional dependencies $F$, written names $(F)$, should be part of the table record type $R$.

$$
\begin{aligned}
& \text { T-Lens } \\
& \frac{(S=\operatorname{Rel}(R, \text { true }, F)) \in \Phi \quad \bigcup \operatorname{names}(F) \subseteq \operatorname{dom}(R)}{\Gamma \vdash \operatorname{lens} S \text { of } R \text { with } F: \text { lens of }(\{S\}, R, \text { true }, F)}
\end{aligned}
$$

We require a relation with the same type to be in the implicit database schema. In practice the implementation trusts that the database contains a table matching the above constraints. This condition is difficult to enforce statically, because the database may be changed after the program is compiled or even during the execution of the program. A good alternative is for the application to perform a check at runtime instead.

### 5.2.2 Select Lens

The select lens takes a predicate $Q$ and filters out records from the view that don't satisfy this predicate.

Section 4.2.4 discusses the ignores constraint. To prevent invalid combination of lenses, the select lens needs to ensure that the underlying lens has no predicate constraints on any fields which may be changed by functional dependencies. The columns affected by functional dependencies $F$ are all the columns outputs $(F)$. It is sufficient to show that the predicate does not refer to any of these columns. Formally, the select lens requires proof that $P$ ignores outputs $(F)$. Lemma 45 shows the property can be derived by showing $R^{\prime} \vdash P$ : bool, where $R^{\prime}$ is equal to $R$ without the columns outputs $(F)$.

The T-Select rule also needs to ensure that the resulting lens only accepts records that satisfy the given predicate $Q$ as well as any existing constraints $P$ that already apply to the underlying lens. The resulting lens constraint predicate can thus be defined as $P \wedge Q$. The full select lens typing rule is:

$$
\begin{aligned}
& \text { T-Select } \\
& \Gamma \vdash e: \text { lens of }(\Sigma, R, P, F) \quad R \vdash Q: \text { bool } \\
& F \text { is in tree form } \quad P \text { ignores outputs }(F) \\
& \hline \Gamma \vdash \operatorname{select}_{Q} \text { from } e: \text { lens of }(\Sigma, R, P \wedge Q, F)
\end{aligned}
$$

### 5.2.3 Join Lens

The join lens performs the natural join of two underlying views. A join lens has limitations on the functional dependencies of the underlying tables. Let us assume that there is another table reviews which contains album reviews by users. The table has the functional dependency user album -> review ${ }^{1}$

[^2]| user | review | album |
| :---: | :---: | :---: |
| musicfan | 4 | Galore |
| 90sclassics | 5 | Galore |
| thecure | 5 | Paris |

The reviews table is joined with the tracks table to produce the lens $l_{1}$. Suppose the user tries to delete the first "90sclassics" record:

| user | review | track | date | rating | album |
| :---: | :---: | :---: | :---: | :---: | :---: |
| musicfan | 4 | Lullaby | 1989 | 3 | Galore |
| musicfan | 4 | Lovesong | 1989 | 5 | Galore |
| 90sclassics | 5 | Lullaby | 1989 | 3 | Galore |
| 90sclassics | 5 | Lovesong | 1989 | 5 | Galore |
| thecure | 5 | Lovesong | 1989 | 5 | Paris |

In this case, there is no way to define a correct behaviour for put. If the user's review is deleted then the other entry by the same user would also be removed from the joined table. If the track is deleted, then the entry from the other user for the same track would also be removed.

The issue is resolved by requiring that one of the tables is completely determined by the join key. The added functional dependency restriction ensures that each entry in the resulting view is associated with exactly one entry in the left table. In this case, if the reviews table contained a single review per track, it would be possible to delete any individual record by only deleting the entry in the reviews table. In practice, we need to show that we can derive the functional dependency $U \cap V \rightarrow V$, where $U \cap V$ are the join columns and $V$ is the set of columns of the right table. We can check if this functional dependency can be derived by calculating the transitive closure of $U \cap V$ and then checking if $V$ is a subset.

Join lenses come in different variants with varying deletion behaviours: a variant that always deletes the entry from the left table, a variant that tries to delete from the right table and otherwise deletes from the left table, and a variant that deletes the entries from both tables if possible. We first look at the delete left variant. The rule T-Join-LEFT requires us to also show that $P$ ignores outputs $(F)$ and $Q$ ignores outputs $(G)$. The resulting lens should have the predicate $P \wedge Q$ since the record constraints of both input lenses apply to the output lens.

| T-Join-LEFT |
| :--- |
| $\Gamma \vdash e:$ lens of $(\Sigma, R, P, F)$ <br> $G \vDash \operatorname{dom}(R) \cap \operatorname{dom}\left(R^{\prime}\right) \rightarrow \operatorname{dom}\left(R^{\prime}\right)$$\quad \Gamma \vdash f$ is in tree form of $\left(\Delta, R^{\prime}, Q, G\right)$ |
| $P$ ignores outputs $(F) \quad Q$ is in tree form |
| $\Gamma \vdash$ join_dl $e$ with $f:$ lens of $\left(\Sigma \cup \Delta, R \cup R^{\prime}, P \wedge Q, F \cup G\right)$ |

Join Template Lens The join template lens can be used to specify the deletion behaviour. The lens requires the programmer to specify two predicates, $P_{d}$ and $Q_{d}$ where $P_{d}$ is a predicate which determines if the predicate should be deleted from the left table and $Q_{d}$ specifies if the record should be deleted from the right table.

An important side condition of the predicates $P_{d}$ and $Q_{d}$ is that any record must either be in $P_{d}$ or $Q_{d}$. If this condition is not met, the lens cannot be well-behaved.

A nicer approach than requiring the predicates to fulfil these side-conditions is to use a method that is correct by construction. Such a method could be for the programmer to specify a total function $\lambda_{d}: R \rightarrow(\nwarrow|\uparrow| \nearrow)$, where $(\nwarrow|\uparrow| \nearrow)$ is a variant type that can either be $\nwarrow, \uparrow$ or $\nearrow$.

The predicate sets $P_{d}$ and $Q_{d}$ are derived from the deletion function $\lambda_{d}$.
Definition 28. Given a deletion function $\lambda_{d}: R \rightarrow(\nwarrow|\uparrow| \nearrow)$, define the deletion functions $P_{d}$ and $Q_{d}$ as:

$$
\begin{aligned}
& P_{d}=\left\{r \mid r \in R, \lambda_{d} r=\nwarrow \text { or } \lambda_{d} r=\uparrow\right\} \\
& Q_{d}=\left\{r \mid r \in R, \lambda_{d} r=\uparrow \text { or } \lambda_{d} r=\nearrow\right\}
\end{aligned}
$$

Lemma 57. Given a deletion function $\lambda_{d}: R \rightarrow(\nwarrow|\uparrow| \nearrow)$, for any record $r: R$ either $r \in P_{d}$ or $r \in Q_{d}$ and therefore $P_{d} \cup Q_{d}=\top_{\operatorname{dom}(R)}$.

Proof.
$\lambda_{d}: R \rightarrow(\nwarrow|\uparrow| \nearrow)$
suppose $\lambda_{d}$
$r: R$
suppose $r$
first show $\forall r: R . r \in P_{d}$ or $r \in Q_{d}$ by destruction on $\lambda_{d} r$ case $\lambda_{d} r=\nwarrow$

$$
\begin{array}{rr}
r & =P_{d} \\
\text { case } \lambda_{d} r=\uparrow & \text { def. } P_{d} \\
& r \in P_{d} \\
\text { case } \lambda_{d} r=\nearrow & \text { def. } P_{d} \\
r & \in Q_{d} \\
P_{1} & \cup Q_{2} \\
& =\left\{r \mid r \in P_{d}\right\} \cup\left\{r \mid r \in Q_{d}\right\} \\
& =\left\{r \mid r \in P_{d} \text { or } r \in Q_{d}\right\} \\
& =\{r \mid r: R\} \\
& =\top_{R}
\end{array}
$$

The typing rule for the template join lens includes an additional premise $\Gamma \vdash$ $f: R \rightarrow(\nwarrow|\uparrow| \nearrow)$, where $f$ represents the deletion predicate $\lambda_{d}$.

## T-Join-Template

$$
\begin{gathered}
\Gamma \vdash e_{1}: \text { lens of }(\Sigma, R, P, F) \\
\Gamma \vdash e_{2}: \text { lens of }\left(\Delta, R^{\prime}, Q, G\right) \quad \Gamma \vdash f: R \cup R^{\prime} \rightarrow(\nwarrow|\uparrow| \nearrow) \\
G \vDash \operatorname{dom}(R) \cap \operatorname{dom}\left(R^{\prime}\right) \rightarrow \operatorname{dom}\left(R^{\prime}\right) \quad F \text { is in tree form } \quad G \text { is in tree form } \\
P \text { ignores outputs }(F) \quad Q \text { ignores outputs }(G) \quad \Sigma \cap \Delta=\varnothing \\
\Gamma \vdash \operatorname{join}_{f} \text { e } e_{1} \text { with } e_{2}: \text { lens of }\left(\Sigma \cup \Delta, R \cup R^{\prime}, P \wedge Q, F \cup G\right)
\end{gathered}
$$

The derived join lenses can be defined using syntactic sugar of the join template lens. For each of the three cases it is easy to show that the $\lambda_{d}$ function terms $\lambda x . \nwarrow, \lambda x . \nearrow$ and $\lambda x . \uparrow$ are total functions of type $R \rightarrow(\nwarrow|\uparrow| \nearrow)$. We use the following definitions:

$$
\begin{aligned}
& \text { join_dl } e \text { with } f=\operatorname{join}_{(\lambda x . \nwarrow)} e \text { with } f \\
& \text { join_dr } e \text { with } f=\operatorname{join}_{(\lambda x . \nearrow)} e \text { with } f \\
& \text { join_db } e \text { with } f=\operatorname{join}_{(\lambda x . \uparrow)} e \text { with } f
\end{aligned}
$$

### 5.2.4 Drop Lens

The drop lens allows a more fine-grained notion of relational projection, removing a column from a view. Note that this is not to be confused with the SQL DROP
statement, which deletes a table. We use the drop term to remain consistent with Bohannon et al. [12].

The term drop $\ell$ determined by $(U, v)$ from $e$ constructs a lens which removes column $\ell$ from view $e$, given that the functional dependencies of the view ensure that $\ell$ is determined by the columns $U$. The typing rule for the drop lens is as follows:

$$
\begin{aligned}
& \text { T-Drop } \\
& \qquad \begin{array}{l}
F \equiv G \cup\{U \rightarrow \ell\} \quad \Gamma \vdash e: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P, F\right) \\
U \subseteq \operatorname{dom}(R) \quad \cdot \vdash v: A
\end{array} \\
& \mathbf{L J D}[R,(\ell: A)](P) \quad \operatorname{DV}[R,(\ell: A)](P,(\ell=v)) \\
& \Gamma \vdash \text { drop } \ell \text { determined by }(U, v) \text { from } e: \text { lens of }\left(\Sigma, R, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right)
\end{aligned}
$$

The clause $F \equiv G \cup\{U \rightarrow \ell\}$ checks that the functional dependencies of the underlying lens $e$ imply that $U$ does indeed determine $\ell$; that $U$ is contained in the domain of the record type $R$ of underlying lens $e$; that $v$ has the same type as the dropped field; that $R$ and ( $\ell: A$ ) define a lossless join decomposition with respect to the lens predicate; and finally that $v$ is a suitable default value with respect to the predicate.

The resulting type lens of $\left(\Sigma, R, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right)$ contains the updated record type without the dropped column, and the updated predicate with the default variable in place of all references to the dropped column.

### 5.2.5 Rename Lens

The rename lens changes the name of column $\ell$ to $\ell^{\prime}$ in the output view. The typing rule requires the provided lens to contain a column $\ell: A$, and replaces it with the column $\ell^{\prime}: A$ in the resulting type. The provided lens may not contain an existing column $\ell^{\prime}$. All references to the column $\ell$ in the predicate $P$ and functional dependencies $F$ are replaced with $\ell^{\prime}$ using substitution. The substitution $\left[\ell / \ell^{\prime}\right]$, defined on both predicates and functional dependencies, replaces all occurrences of $\ell$ with $\ell^{\prime}$. The semantics of substitution is straightforward and is not further described.

T-REnAmE

$$
\frac{\Gamma \vdash e: \text { lens of }(\Sigma, R \oplus(\ell: A), P, F)}{\Gamma \vdash \operatorname{rename}_{\ell / \ell^{\prime}} \text { from } e: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P\left[\ell / \ell^{\prime}\right], F\left[\ell / \ell^{\prime}\right]\right)}
$$

### 5.2.6 Lens Functions

Lens Get Finally, we define typing rules for making use of relational lenses. Since the target language is not dependently typed, we discard the constraints which apply to the view, and specify that calling get returns a set of records which all have the type $R$.

$$
\begin{aligned}
& \text { T-GET } \\
& \frac{\Gamma \vdash e: \text { lens of }(\Sigma, R, P, F)}{\Gamma \vdash \text { get } e:\{R\}}
\end{aligned}
$$

Lens Put Just as with T-Get, we have no way of statically ensuring that the input satisfies $P$ and $F$, so we only statically check that the updated view is a set of records matching type $R$, deferring to runtime the checks to ensure that the set of records satisfies $F$ and $P$.

To ensure that the constraint $P$ applies to each record $r$ in a view, runtime checks ensure that sat $(P, r)$. Functional dependency constraints can be checked by projecting the set of records down to each functional dependency and determining if any two records violate a functional dependency.

The result of calling put indicates whether the input view computed by $f$ satisfies the constraints $P$ and $F$ and hence the database could be updated.

$$
\begin{aligned}
& \text { T-Put } \\
& \frac{\Gamma \vdash e: \text { lens of }(\Sigma, R, P, F) \quad \Gamma \vdash f:\{R\}}{\Gamma \vdash \text { put } e \text { with } f: \text { bool }}
\end{aligned}
$$

### 5.3 Evaluation Rules

Figure 5.5 introduces the big-step evaluation rules of our sequential lenses. The evaluation rule $e, \varphi \Downarrow v, \gamma$ evaluates the term $e$ given a database state $\varphi$ and yields the value $v$, while changing the database state to $\gamma$. We assume the rules

$$
\begin{aligned}
& \text { E-Value E-App } \\
& \begin{array}{lll}
\overline{v, \varphi \Downarrow v, \varphi} \quad \begin{array}{ll}
e, \varphi \Downarrow \lambda x \cdot e_{1}, \varphi_{1} & f, \varphi_{1} \Downarrow v, \varphi_{2}
\end{array} \quad e_{1}[v / x], \varphi_{2} \Downarrow w, \gamma \\
e f, \varphi \Downarrow w, \gamma
\end{array} \\
& \begin{array}{ll}
\begin{array}{l}
\text { E-RECORD } \\
\left(e_{i}, \varphi_{i} \Downarrow v_{i}, \varphi_{i+1}\right)_{i \in 1 \ldots n}
\end{array} & \begin{array}{l}
\text { E-Const-SET } \\
(\overrightarrow{\ell=e}), \varphi_{1} \Downarrow(\overrightarrow{\ell=v}), \varphi_{n+1}
\end{array}
\end{array} \quad \frac{\left(e_{i}, \varphi_{i} \Downarrow v_{i}, \varphi_{i+1}\right)_{i \in 1 \ldots n}}{\{\vec{e}\}, \varphi_{1} \Downarrow\{\vec{w}\}, \varphi_{n+1}} \quad \vec{w}=\text { set of } v \\
& \text { E-Lens-Select } \\
& \text { E-Join } \\
& e_{1}, \varphi \Downarrow v_{1}, \varphi_{1} \quad e_{2}, \varphi_{1} \Downarrow v_{2}, \varphi_{2} \quad f, \varphi_{2} \Downarrow w, \gamma \\
& \operatorname{join}_{f} e_{1} \text { with } e_{2}, \varphi \Downarrow \text { join }_{w} v_{1} \text { with } v_{2}, \gamma
\end{aligned}
$$

E-Drop

$$
e, \varphi \Downarrow w, \gamma
$$

drop $\ell$ determined by $(U, v)$ from $e, \varphi \Downarrow \operatorname{drop} \ell$ determined by $(U, v)$ from $w, \gamma$

E-Rename
$\frac{e, \varphi \Downarrow v, \gamma}{\operatorname{rename}_{\ell / \ell^{\prime}} \text { from } e, \varphi \Downarrow \text { rename }_{\ell / \ell^{\prime}} \text { from } v, \gamma} \quad \frac{e, \varphi \Downarrow v, \gamma \quad w=\operatorname{get}_{v}(\gamma)}{\text { get } e, \varphi \Downarrow w, \gamma}$

E-Put-Sat
$e, \varphi \Downarrow v, \varphi_{1} \quad f, \varphi_{1} \Downarrow w, \varphi_{2} \quad v:$ lens of $(\Sigma, S, P, F)$
$w \models F \quad \forall r \in w . \operatorname{sat}(P, r)$

$$
\gamma=p u t_{\varphi_{2}, v}(w)
$$

put $e$ with $f, \varphi \Downarrow$ true, $\gamma$
E-Put-Unsat

$$
\begin{aligned}
& e, \varphi \Downarrow v, \varphi_{1} \quad f, \varphi \Downarrow w, \gamma \quad v: \text { lens of }(\Sigma, S, P, F) \\
& w \not \models F \text { or } \exists r \in w . \neg \operatorname{sat}(P, r) \\
& \text { put } e \text { with } f, \varphi \Downarrow \text { false, } \gamma
\end{aligned}
$$

Figure 5.5: Lens evaluation rules.

E-Value and E-App, which are standard big-step evaluation rules. The ERECORD rule evaluates all the subterms and constructs the corresponding record. E-Const-Set evaluates all the arguments and constructs a set from them. The primitive lens is always a value, and therefore covered by the E-VALUE rule. The join, drop, select and rename lenses all require the subterms to be evaluated first and the resulting values inserted into the lens constructor. As the join variants are definable as syntactic sugar, the join lens variants are all handled by the E-Join rule.

In the E-Get rule, we assume that $\operatorname{get}_{v}(\varphi)$ queries the database server and returns the resulting view. The put term has two evaluation outcomes. If the input view satisfies the functional dependency and predicate constraints, then the E-Put-Sat rule returns the new database state $\gamma=\operatorname{put}_{v}\left(\varphi_{2}, w\right)$, where $\varphi_{2}$ is the database state after evaluating all the subterms. The E-Put-Sat rule then yields the value true, indicating that the operation was successful. If the view does not satisfy the predicate and functional dependency constraints, then the E-Put-Unsat rule applies, which returns false indicating the operation failed.

### 5.4 Translation

Bohannon et al. [12] prove that lenses satisfying correctness conditions are wellbehaved (i.e., satisfy GetPut and PutGet, and therefore safely compose). Their typing rules are not in a form amenable to implementation, since predicates are defined as abstract sets; lenses are composed using a sequential composition operator as defined in Section 2.2 rather than allowing arbitrarily-nested lenses as one would in a functional language; and there is no distinction between a relation and a lens on a relation.

Nevertheless, we must show that our typing rules also guarantee well-behavedness. Our approach is to define a type-preserving translation from our functional-style lenses into the sequential-style lenses defined by Bohannon et al. [12].

Figure 5.6 shows the translation from functional lenses to sequential-style lenses. The process involves flattening functional lenses by introducing intermediate relations with fresh table names. The translation function $(v)=\Sigma / I / S$ states that functional lens value $v$ depends on tables $\Sigma$, translates to sequential lens $I$, and produces a view with name $S$.

## Flattening translation

```
(lens S of R with F) ={S}/id/S
(select }\mp@subsup{P}{P}{\mathrm{ from v}
    \Sigma/I; select tet (P,\operatorname{dom}(S))}S\mathrm{ as T/T
    where (v) =\Sigma/I/S and T is globally unique
(join_dl v with wl)=
    \Sigma\uplus\Delta/I; J;join_dl S S S S as T/T
    where (v) =\Sigma/I/S S,(w) = \Delta/J/S S and T is globally unique
(drop \ell determined by (V,v) from w) =
    \Sigma/I;drop \ell determined by ( }V,v)\mathrm{ from S as T/T
    where (w) =\Sigma/I/S
```

Figure 5.6: Sequential-style lenses [12] and flattening

We would like to show that any well-typed closed lens value $v$ translates to a well-behaved sequential lens expression $I$. We require that any lens expression $v$ such that $\cdot \vdash v$ lens of $(\Sigma, R, P, F)$ and $(v)=\Delta / I /\{T\}$, then $\Sigma=\Delta, I \in \Sigma \Leftrightarrow$ $\{T\}$ and sort $(T)=(R, P, F)$. We first show this property holds true for all lens constructors. The properties that should be satisfied are also the invariants we use for each sub-term.

Lemma 58 (Sequential Lens Weakening). Given $\Sigma$ and $\Delta$ which are disjoint from $\Sigma^{\prime}$. If $I \in \Sigma \Leftrightarrow \Delta$, and all intermediate views in I are globally unique, then $I \in \Sigma \uplus \Sigma^{\prime} \Leftrightarrow \Delta \uplus \Sigma^{\prime}$.

Proof. For select, join and projection lenses of type $\Sigma \Leftrightarrow \Delta$ the typing rules allows us to extend both schemas in the lens type with additional relation names as long as they are disjoint from $\Sigma$ and $\Delta$. If the lens $I$ is a sequential composition of two lenses of types $\Sigma \Leftrightarrow \Delta^{\prime}$ and $\Delta^{\prime} \Leftrightarrow \Delta$, then the global uniqueness conditions ensures that $\Delta^{\prime}$ is disjoint from $\Sigma^{\prime}$. We apply the induction hypothesis to both lenses, and then use the sequential composition rule.

## Primitive Lens

Sequential relational lenses are not actually able to introduce completely new views in the schema. Instead they assume that underlying tables exist, and so
the primitive lens for a table $S$ is actually just translated into an identity lens, while asserting that the table $S$ exists in the schema.

Lemma 59. Suppose $\cdot \vdash$ lens $S$ of $R$ with $F$ :lens of $(\Sigma, R$, true,$F)$. Then $($ lens $S$ of $R$ with $F)=\{S\} / \boldsymbol{i d} / S, \quad$ id $\in\{S\} \Leftrightarrow\{S\} \quad$ and $\operatorname{sort}(S)=$ $(\operatorname{dom}(R), \operatorname{set}(\boldsymbol{t r u e}, R), F)$.

Proof.
T-Lens-Prim
$(S=\operatorname{Rel}(R$, true,$F)) \in \Phi \quad \bigcup \operatorname{names}(F) \subseteq \operatorname{dom}(R)$
$\Gamma \vdash$ lens $S$ of $R$ with $F:$ lens of $(\{S\}, R$, true, $F)$
$\operatorname{sort}(S)=(\operatorname{dom}(R), \operatorname{set}(\operatorname{true}, R), F) \quad$ define
$I=\mathbf{i d} \in\{S\} \Leftrightarrow\{S\}$ T-Id-RL
(lens $e$ with $F$ ) $=\{S\} / I / S$

## Select Lens

The select lens filters out records that do not match a user supplied predicate. Rather than taking an abstract predicate, the select lens in our language must take a concrete predicate. We assume a predicate in the basic predicate syntax introduced in Section 4.1.

The select lens translates into a lens composition $I ; J$, where $J$ is a select lens and $I$ is the lens computed from the expression $e$. The select lens typing rule requires us to relate the logical and-operator with set intersection on set predicates, which has been covered in Section 4.2.1. It is also necessary to show that the ignores constraint on the concrete predicate implies the ignores constraint on the abstract set, which is covered in Section 4.2.4.

Lemma 60. Suppose $\cdot \vdash v$ :lens of $(\Sigma, R, P, F)$, $\vdash$ select $_{Q}$ from $v$ : lens of $(\Sigma, R, P \wedge Q, F)$ and $(v)=\Sigma / I / S$ such that $I \in \Sigma \Leftrightarrow\{S\}$ and $\operatorname{sort}(S)=$ $(\operatorname{dom}(R), \operatorname{set}(P, R), F)$. Then $\left(\right.$ select $_{Q}$ from $\left.v\right)=\Sigma / I^{\prime} / T$ such that $I^{\prime} \in \Sigma \Leftrightarrow$ $\{T\}$ and $\operatorname{sort}(T)=(\operatorname{dom}(R), \operatorname{set}(P \wedge Q, R), F)$.

```
Proof.
    \(\cdot \vdash v\) :lens of \((\Sigma, R, P, F) \quad R \vdash Q\) : bool
\(F\) is in tree form \(\quad P\) ignores outputs \((F)\)
\(\cdot \vdash\) select \(_{Q}\) from \(v\) :lens of \((\Sigma, R, P \wedge Q, F)\)
\((v)=\Sigma / I / S\)
\(I \in \Sigma \Leftrightarrow\{S\}\)
\(\operatorname{sort}(S)=(\operatorname{dom}(R), \operatorname{set}(P, R), F)\)
\(\operatorname{sort}(T)=(\operatorname{dom}(R), \operatorname{set}(P \wedge Q, R), F)\)
    \(=(\operatorname{dom}(R), \operatorname{set}(P, R) \cap \operatorname{set}(Q, R), F)\)
\(\operatorname{set}(Q, R)\) ignores outputs \((F)\)
\(J=\) select from \(S\) where \(\operatorname{set}(Q, R)\) as \(T\)
    \(\in\{S\} \Leftrightarrow\{T\}\)
\(\left(\right.\) select \(_{Q}\) from \(v\) ) \(=\Sigma / I ; J / T\)
\(I ; J \in \Sigma \Leftrightarrow\{T\}\)
```

assumption (1)
assumption (2)
assumption (3)
assumption (4)
T-Select-RL 4677
def. (.)

T-Compose-RL (38)

## Join Lens

The join lens performs a natural join between the two input views. The template join lens allows the programmer to specify a function $f: R \rightarrow(\nwarrow|\uparrow| \nearrow)$ that determines which table to delete any record $r$ such that $r: R$ from, if deleting the record from either table would correctly propagate the changes. The delete left, delete right and delete both join table variants can be derived from the template join using the corresponding deletion function.

We show that our join lens can derive a correct sequential style relational lens. The corresponding sequential lens is computed by first composing the translation of the two input lens expressions $e_{1}$ and $e_{2}$, and then composing a join lens on top of that. We rely on Lemma 57 to show that the equivalent deletion predicates $P_{d}$ and $Q_{d}$ satisfy the requirements necessary. As with the select lens, work in Section 4.2.4 shows that the ignores requirement on the predicate set constraints of each lens can be derived from the ignores constraint on basic predicate.

Lemma 61. Suppose $\cdot \vdash v_{1}:$ lens of $(\Sigma, R, P, F), \cdot \vdash v_{2}:$ lens of $\left(\Delta, R^{\prime}, Q, G\right)$, $\cdot \vdash \boldsymbol{j o i n}_{f} v_{1}$ with $v_{2}:$ lens of $\left(\Sigma \uplus \Delta, R \cup R^{\prime}, P \cap Q, F \cup G\right)$ and $\left(v_{1}\right)=\Sigma / I_{1} / S_{1}$
and $\left(v_{2}\right)=\Delta / I_{2} / S_{2} \quad$ such that $I_{1} \in \Sigma \Leftrightarrow\left\{S_{1}\right\}$, $\operatorname{sort}\left(S_{1}\right)=$ $(\operatorname{dom}(R), \operatorname{set}(P, R), F)$ and $I_{2} \in \Delta \Leftrightarrow\left\{S_{2}\right\}$, $\operatorname{sort}\left(S_{2}\right)=\left(\operatorname{dom}\left(R^{\prime}\right), \operatorname{set}\left(Q, R^{\prime}\right), G\right)$. Then $\left(\boldsymbol{j o i n}_{w} v_{1}\right.$ with $\left.v_{2}\right)=\Sigma \uplus \Delta / I / T$ such that $I \in \Sigma \uplus \Delta \Leftrightarrow\{T\}$ and $\operatorname{sort}(T)=$ $\left(\operatorname{dom}\left(R \oplus R^{\prime}\right), \operatorname{set}\left(P \cap Q, R \oplus R^{\prime}\right), F \cup G\right)$.

Proof.
T-Join-Template

| $\cdot \vdash v_{1}: \operatorname{lens}$ of $(\Sigma, R, P, F)$ | $\cdot \vdash v_{2}:$ lens of $\left(\Delta, R^{\prime}, Q, G\right)$ | $\cdot \vdash w: R \rightarrow(\nwarrow\|\uparrow\| \nearrow)$ |
| :---: | :---: | :---: |
| $G \vDash \operatorname{dom}(R) \cap \operatorname{dom}\left(R^{\prime}\right) \rightarrow \operatorname{dom}\left(R^{\prime}\right) \quad F$ is in tree form | $G$ is in tree form |  |
| $P$ ignores outputs $(F) \quad Q$ ignores outputs $(G)$ | $\Sigma \cap \Delta=\varnothing$ |  |
| $\vdash \operatorname{join}_{w} v_{1}$ with $v_{2}:$ lens of $\left(\Sigma \cup \Delta, R \cup R^{\prime}, P \cap Q, F \cup G\right)$ |  |  | assumption

$\left(v_{1}\right)=\Sigma / I_{1} / S_{1}$
$\left(v_{2}\right)=\Sigma / I_{2} / S_{2}$
$I_{1} \in \Sigma \Leftrightarrow\left\{S_{1}\right\}$
$I_{2} \in \Delta \Leftrightarrow\left\{S_{2}\right\}$
$\operatorname{sort}\left(S_{1}\right)=(\operatorname{dom}(R), \operatorname{set}(P, R), F)$
mption
$\operatorname{sort}\left(S_{2}\right)=\left(\operatorname{dom}\left(R^{\prime}\right), \operatorname{set}\left(Q, R^{\prime}\right), G\right)$
assumption
$P_{d}=\left\{r \mid r \in R, \lambda_{d} r=\nwarrow\right.$ or $\left.\lambda_{d} r=\uparrow\right\}$
$Q_{d}=\left\{r \mid r \in R, \lambda_{d} r=\uparrow\right.$ or $\left.\lambda_{d} r=\nearrow\right\}$
$P_{d} \cup Q_{d}=\top_{\operatorname{dom}\left(R \oplus R^{\prime}\right)}$
Lemma 57
$I_{1} \in \Sigma \uplus \Delta \Leftrightarrow\left\{S_{1}\right\} \uplus \Delta$
weakening*
$I_{2} \in\left\{S_{1}\right\} \uplus \Delta \Leftrightarrow\left\{S_{1}, S_{2}\right\}$
weakening*
$I_{1} ; I_{2} \in \Sigma \uplus \Delta \Leftrightarrow\left\{S_{1}, S_{2}\right\}$
T-Compose-RL

* all intermediate views are globally unique due to def. of (.).
sort ( $T$ )

$$
\begin{align*}
& =\left(\operatorname{dom}\left(R \oplus R^{\prime}\right), \operatorname{set}\left(P \wedge Q, R \oplus R^{\prime}\right), F \cup G\right)  \tag{7}\\
& =\left(\operatorname{dom}(R) \cup \operatorname{dom}\left(R^{\prime}\right), \operatorname{set}(P, R) \bowtie \operatorname{set}\left(Q, R^{\prime}\right), F \cup G\right)
\end{align*}
$$

define
Lemma 39 (8)
$\operatorname{set}(P, R)$ ignores outputs $(F)$
$\operatorname{set}\left(Q, R^{\prime}\right)$ ignores outputs $(G)$
Set-Ignores (1)
SET-IGNores (1) (10)

T-Join-RL
$J=\operatorname{join}_{P_{d}, Q_{d}} S_{1} S_{2}$ as $T \in\left\{S_{1}, S_{2}\right\} \Leftrightarrow\{T\}$
$\left(\right.$ join $_{w} v_{1}$ with $\left.v_{2}\right)=\Sigma \uplus \Delta / I_{1} ; I_{2} ; J / T$
$I_{1} ; I_{2} ; J \in \Sigma \uplus \Delta \Leftrightarrow\{T\}$
T-Compose-RL

## Drop Lens

The drop lens projects away some columns from the view. The drop lens must show that it is safe to remove the portion of the predicate that refers to the dropped columns, as well as that the default value to be used for the dropped columns satisfies those dropped portions of the predicate. These restrictions, known as the lossless join decomposition and default value checks, are performed on the basic predicate constraint of the underlying lens refinement type.

Based on the work in Section 4.2 we can show that performing these checks on basic predicates implies that the equivalent set predicates satisfy these conditions as well. Just as with the select lens, the drop lens translates into a composition of the translated underlying lens and a sequential drop lens.

Lemma 62. Suppose . $\vdash$ w: lens of $(\Sigma, R, P, F)$ as well as $\cdot \vdash$ drop $\ell$ determined by $(U, v)$ from $w: l e n s$ of $\left(\Sigma, R^{\prime}, Q, G\right)$ and $(w)=\Sigma / I / S$ such that $I \in \Sigma \Leftrightarrow\{S\}$ and $\operatorname{sort}(S)=(\operatorname{dom}(R), \operatorname{set}(P, R), F)$. Then (drop $\ell$ determined by $(U, v)$ from $w)=\Sigma / J / T$ such that $J \in \Sigma \Leftrightarrow\{T\}$ and $\operatorname{sort}(T)=\left(\operatorname{dom}\left(R^{\prime}\right), \operatorname{set}(Q, R), G\right)$.

Proof.
T-Drop

$$
\begin{gather*}
F \equiv G \cup\{U \rightarrow \ell\} \quad w \vdash \text { lens of }(\Sigma, R \oplus(\ell: A), P, F): \\
U \subseteq \operatorname{dom}(R) \quad \cdot \vdash v: A \\
\mathbf{L J D}[R,(\ell: A)](P) \quad \operatorname{DV}[R,(\ell: A)](P,(\ell=v)) \tag{1}
\end{gather*}
$$

$\cdot \vdash$ drop $\ell$ determined by $(U, v)$ from $w:$ lens of $\left(\Sigma, R, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right)$

$$
\begin{array}{ll}
(w)=\Sigma / I / S & \text { assumption (2) } \\
I \in \Sigma \Leftrightarrow\{S\} & \text { assumption (3) } \\
\begin{aligned}
\operatorname{sort}(S)=(\operatorname{dom}(R \oplus(\ell: A)), \operatorname{set}(P, R \oplus(\ell: A)), F) & \text { assumption } \\
\quad=(\operatorname{dom}(R) \cup\{\ell\}, \operatorname{set}(P, R \oplus(\ell: A)), F) & \text { assumption (4) } \\
& \\
\operatorname{set}(P, R \oplus(\ell: A)) & \\
\quad=\operatorname{set}(P, R \oplus(\ell: A))[\operatorname{dom}(R)] \bowtie \operatorname{set}(P, R \oplus(\ell: A))[\ell] & \text { Lemma40 (5) } \\
(\ell=v) \in \operatorname{set}\left(P\left[\ell^{\prime}\right], R\right) & \text { Lemma 42 (6) }
\end{aligned}
\end{array}
$$

```
\(\operatorname{sort}(T)=\left(\operatorname{dom}(R), \operatorname{set}\left(\llbracket P \rrbracket_{R,(\ell: A)}, R\right), G\right)\)
    \(=(\operatorname{dom}(R), \operatorname{set}(P, R \oplus(\ell: A))[\operatorname{dom}(R)], G)\)
```

define Lemma 49 (7)

* for the special case where $\operatorname{set}(P, R \oplus(\ell: A))=\varnothing$ see Appendix D. 3
$\{\ell\} \in \operatorname{dom}(R) \cup\{\ell\}$
trivial (8)

```
\(J=\) drop \(\ell\) determined by \((U, v)\) from \(S\) as \(T\)
    \(\in\{S\} \Leftrightarrow\{T\} \quad\) T-Drop-RL \(4|8| 1775 \mid 6\) (9)
\(w^{\prime}=\operatorname{drop} \ell\) determined by \((U, v)\) from \(w\)
```

(drop $\ell$ determined by $(U, v)$ from $w)=\Sigma / I ; J / T$
$I ; J \in \Sigma \Leftrightarrow\{T\}$

T-Drop-RL (4) 8 |17|5|6) (9)
define
(2); def. ( $\cdot$ )

T-Compose-RL (39)

## Rename Lens

The rename lens requires us to first relate renaming in functional predicates to the rename operator on predicate sets.

Lemma 63. Given $R$ disjoint from $\left\{\ell, \ell^{\prime}\right\}$ and $R \oplus(\ell: A) \vdash P:$ bool then

$$
\operatorname{set}\left(P\left[\ell / \ell^{\prime}\right], R \oplus\left(\ell^{\prime}: A\right)\right)=\rho_{\ell / \ell^{\prime}}(\operatorname{set}(P, R \oplus(\ell: A))) .
$$

Proof.
$R$ disjoint from $\left\{\ell, \ell^{\prime}\right\}$
assumption (1)
$R \oplus \ell: A \vdash P$ : bool
assumption (2)

$$
\begin{array}{rlr}
\operatorname{set} & \left(P\left[\ell / \ell^{\prime}\right], R \oplus\left(\ell^{\prime}: A\right)\right) & \operatorname{def.} \operatorname{set}(\cdot, \cdot) \\
& =\left\{r \in \operatorname{inh}\left(R \oplus\left(\ell^{\prime}: A\right)\right) \mid \operatorname{sat}\left(P\left[\ell / \ell^{\prime}\right], r\right)\right\} & \text { 11 } ; \operatorname{def.} \operatorname{inh}(\cdot) \\
& =\left\{r\left[\ell / \ell^{\prime}\right] \in \operatorname{inh}(R \oplus(\ell: A)) \mid \operatorname{sat}\left(P\left[\ell / \ell^{\prime}\right], r\left[\ell / \ell^{\prime}\right]\right)\right\} & \text { 11,22) } \\
& =\left\{r\left[\ell / \ell^{\prime}\right] \in \operatorname{inh}(R \oplus(\ell: A)) \mid \operatorname{sat}(P, r)\right\} & \operatorname{def} . \rho \cdot(\cdot) \\
& =\rho_{\ell / \ell^{\prime}}(\{r \in \operatorname{inh}(R \oplus(\ell: A)) \mid \operatorname{sat}(P, r)\}) & \operatorname{def} . \operatorname{set}(\cdot, \cdot)
\end{array}
$$

Lemma 64. Suppose $\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$, $\vdash$ rename $_{\ell / \ell^{\prime}}$ from $v$ : lens of $\left(\Sigma, R^{\prime}, Q, G\right)$ and $(v)=\Sigma / I / S$ such that $I \in \Sigma \Leftrightarrow\{S\}$ and $\operatorname{sort}(S)=$ $(\operatorname{dom}(R), \operatorname{set}(P, R), F)$. Then $\left(\right.$ rename $_{\ell / \ell^{\prime}}$ from $\left.v\right)=\Sigma / J / T$ such that $J \in \Sigma \Leftrightarrow$ $\{T\}$ and $\operatorname{sort}(T)=\left(\operatorname{dom}\left(R^{\prime}\right), \operatorname{set}\left(Q, R^{\prime}\right), G\right)$.

Proof.
$R_{1}=R \oplus(\ell: A)$
$R_{2}=R \oplus\left(\ell^{\prime}: A\right)$
define (2)
$\frac{\cdot \vdash v: \text { lens of }\left(\Sigma, R_{1}, P, F\right)}{\cdot \vdash \text { rename }_{\ell / \ell^{\prime}} \text { from } v: \text { lens of }\left(\Sigma, R_{2}, P\left[\ell / \ell^{\prime}\right], F\left[\ell / \ell^{\prime}\right]\right)}$

| $(v\rangle)=\Sigma / I / S$ | assumption (4) |
| :--- | ---: |
| $I \in \Sigma \Leftrightarrow\{S\}$ | assumption |
| sort $(S)=\left(\operatorname{dom}\left(R_{1}\right), \operatorname{set}\left(P, R_{1}\right), F\right)$ | assumption |
| $\quad=\left(\operatorname{dom}(R) \cup\{\ell\}, \operatorname{set}\left(P, R_{1}\right), F\right)$ | def. $R_{1}(5)$ |
| $\operatorname{sort}(T)=\left(\operatorname{dom}\left(R_{2}\right), \operatorname{set}\left(P\left[\ell / \ell^{\prime}\right], R_{2}\right), F\left[\ell / \ell^{\prime}\right]\right)$ | define |
|  | $=\left(\operatorname{dom}(R) \cup\left\{\ell^{\prime}\right\}, \rho_{\ell / \ell^{\prime}}\left(\operatorname{set}\left(P, R_{1}\right)\right), F\left[\ell / \ell^{\prime}\right]\right)$ |

$\operatorname{dom}(R) \cap\left\{\ell, \ell^{\prime}\right\}=\varnothing$

1. 2]; def. $\oplus(7)$
$J=$ rename $_{\ell / \ell^{\prime}} S$ as $T \in\{S\} \Leftrightarrow\{T\}$
$\left(\right.$ rename $_{\ell / \ell^{\prime}}$ from $\left.v\right)=\Sigma / I ; J / T$
$I ; J \in \Sigma \Leftrightarrow\{T\}$
T-Rename-RL (7, 5, 6)
(4); def. (.)

T-Compose-RL

## Soundness of Lenses

We can now state our soundness theorem, stating that once translated, lenses typeable in our system are typeable using the original rules proposed by Bohannon et al. [12]. All of the above Lemmas are then be combined to show that any welltyped lens expression always produces a well-typed lens.

Theorem 11 (Soundness of Translation).
If $\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$ then $(v)=\Sigma / I / T, I \in \Sigma \Leftrightarrow\{T\}$ and $\operatorname{sort}(T)$ $=(\operatorname{dom}(R), \operatorname{set}(P, R), F)$.

Proof.
$\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$

Perform induction on $\cdot \vdash v$ :lens of $(\Sigma, R, P, F)$

```
\(\cdot \vdash\) lens \(S\) of \(R\) with \(F\) :lens of \((\{S\}, R\), true,\(F)\)
    (lens \(S\) of \(R\) with \(F)=\{S\} / I / S\)
    \(I \in\{S\} \Leftrightarrow\{S\}\)
    \(\operatorname{sort}(S)=(\operatorname{dom}(R), \operatorname{set}(P, R), F)\)
\(\cdot \vdash \operatorname{select}_{Q}\) from \(w\) :lens of \((\Sigma, R, P \wedge Q, F)\)
    \(\cdot \vdash w\) lens of \((\Sigma, R, P, F)\)
    \(\left(\right.\) select \(_{Q}\) from \(\left.w\right)=\Sigma / I / S\)
    \(I \in \Sigma \Leftrightarrow\{S\}\)
    \(\operatorname{sort}(S)=(\operatorname{dom}(R), \operatorname{set}(P, R), F)\)
    \((v)=\Sigma / J / T\)
    \(J \in \Sigma \Leftrightarrow\{T\}\)
    \(\operatorname{sort}(T)=(\operatorname{dom}(R), \operatorname{set}(P \wedge Q, R), F)\)
\(\cdot \vdash \operatorname{join}_{w} v_{1}\) with \(v_{2}:\) lens of \(\left(\Sigma \uplus \Delta, R \cup R^{\prime}, P \wedge Q, F \cup G\right)\)
    \(\cdot \vdash v_{1}\) :lens of \((\Sigma, R, P, F)\)
    \(\cdot \vdash v_{2}\) : lens of \(\left(\Delta, R^{\prime}, Q, G\right)\)
    \(\left(v_{1}\right)=\Sigma / I_{1} / S_{1}\)
    \(I_{1} \in \Sigma \Leftrightarrow S_{1}\)
    \(\operatorname{sort}\left(S_{1}\right)=(\operatorname{dom}(R), \operatorname{set}(P, R), F)\)
    \(\left(v_{2}\right)=\Delta / I_{2} /\left\{S_{2}\right\}\)
    \(I_{2} \in \Delta \Leftrightarrow\left\{S_{2}\right\}\)
    \(\operatorname{sort}\left(S_{2}\right)=\left(\operatorname{dom}\left(R^{\prime}\right), \operatorname{set}\left(Q, R^{\prime}\right), G\right)\)
    \(\left(\mathbf{j o i n}_{w} v_{1}\right.\) with \(\left.v_{2}\right)=\Sigma \uplus \Delta / J / T\)
    \(J \in \Sigma \uplus \Delta \Leftrightarrow\{T\}\)
    \(\operatorname{sort}(T)=\left(\operatorname{dom}\left(R \oplus R^{\prime}\right), \operatorname{set}\left(P \wedge Q, R^{\prime}\right), F \cup G\right)\)
Lemma 61 (5, 6, 7, 8, 9, 10)
Lemma 61 (5) 6, 7, 8, 9, 10)
Lemma 61 (5, 6, 7, 8, 9, 10)
\(\cdot \vdash\) drop \(\ell\) determined by \(\left(U, v^{\prime}\right)\) from \(w\) :lens of \((\Sigma, R, P, F) \quad\) assumption
    \(\cdot \vdash w:\) lens of \(\left(\Sigma, R^{\prime}, Q, G\right)\)
    \((w)=\Sigma / I / T\)
    \(I \in \Sigma \Leftrightarrow\{T\}\)
    \(\operatorname{sort}(S)=\left(\operatorname{dom}\left(R^{\prime}\right), \operatorname{set}\left(Q, R^{\prime}\right), G\right)\)
    T-Drop (11)
    def. ( \(\cdot\) )
by induction (12)
```

T-Drop (11)
def. ( $\cdot$ ) by induction (12) by induction (13)

```
    (drop \ell determined by (U,v') from w) =\Sigma/J/T
J\in\Sigma\Leftrightarrow{T}
sort}(T)=(\operatorname{dom}(R),\operatorname{set}(P,R),F
- }+\mp@subsup{\mathrm{ rename}}{\ell/\mp@subsup{\ell}{}{\prime}}{\mathrm{ from }w:lens of (\Sigma,R,P,F)
    \bullet }\vdash:lens of (\Sigma, R',Q,G
    (w) =\Sigma/I/S
    I\in\Sigma\Leftrightarrow{T}
    sort}(S)=(\operatorname{dom}(\mp@subsup{R}{}{\prime}),\operatorname{set}(Q,\mp@subsup{R}{}{\prime}),G
    (rename }\mp@subsup{\ell/l/\mp@subsup{\ell}{}{\prime}}{\mathrm{ from }w)=\Sigma/J/T}{
J\in\Sigma\Leftrightarrow{T}
sort(T)=(dom(R),\operatorname{set}(P,R),F)
Lemma 62 (11, 12, 13)
Lemma 62 (11, 12, 13)
Lemma 62 11, 12, 13)
Lemma 62 (11, 12, 13)
Lemma 64 (14, 15, 16)
Lemma 64 (14, 15, 16
Lemma 64 (14, 15, 16)
```


### 5.5 Language is well-behaved

We would now like to show that our language behaves well at runtime. This chapter is focused on the typing of relational lenses and does not directly specify the semantics of the lenses themselves. Instead, we rely on the get $_{v}$ and $p u t_{v}$ functions to satisfy certain properties. Appendix D.2 provides an example of non-incremental definitions for the get and put functions, along with proofs they satisfy the specification. We first require $g e t_{v}$ and $p u t_{v}$ to be well-typed total functions:

Lemma 65 (Get Total). If $\varphi: \Phi$ and $\cdot \vdash v$ : lens of $(\Sigma, R, P, F)$ then $\operatorname{get}_{v}(\varphi)=M$ and $M: \operatorname{Rel}(R, P, F)$.

Lemma 66 (Put Total). Given $\varphi: \Phi$, if $M: \operatorname{Rel}(R, P, F)$ and $\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$ then $\operatorname{put}_{v}(\varphi, M)=\gamma$ and $\gamma: \Phi$.

The following lemma states that substitution in our language behaves correctly. A substitution $e[v / x]$ specifies that all free variables $x$ in the term $e$ should be substituted with the value $v$. The definition for substitution is standard and is included in Appendix D. 1 .

Lemma 67 (Substitution). If $\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash e: \tau$ and $\cdot \vdash v: \tau_{1}$ then $\Gamma \vdash e[v / x]: \tau$.

Proof. Shown in Appendix D.1.

Any term that can be evaluated to a value preserves the type of the initial term.
Theorem 12 (Preservation). Given $\varphi: \Phi$, if $\cdot \vdash e: \tau$ and $e, \varphi \Downarrow v, \gamma$ then $\cdot \vdash v: \tau$ and $\gamma: \Phi$.

Proof.
$\varphi: \Phi$ assumption
$\cdot \vdash e: \tau \quad$ assumption (1)
$e, \varphi \Downarrow v, \gamma$
assumption (2)
Perform induction on $e \Downarrow v$
case E-Lens-Select

| $e, \varphi \Downarrow v, \gamma$ | assumption |
| :--- | ---: |
| $\operatorname{select}_{P}$ from $e, \varphi \Downarrow \operatorname{select}_{P}$ from $v, \gamma$ |  |
| $\Gamma \vdash e:$ lens of $(\Sigma, R, P, F) \quad R \vdash Q:$ bool |  |
| $\frac{F \text { is in tree form } \quad P \text { ignores outputs }(F)}{\Gamma \vdash \operatorname{select}_{Q} \text { from } e: \text { lens of }(\Sigma, R, P \wedge Q, F)}$ | T-SELECT |
| $\Gamma \vdash v:$ lens of $(\Sigma, R, P, F)^{\gamma: \Phi}$ | induction |
| $\Gamma \vdash \operatorname{select}_{Q}$ from $v:$ lens of $(\Sigma, R, P \wedge Q, F)$ | induction |

case E-Get

$$
\underline{e^{\prime}, \varphi \Downarrow v^{\prime}, \gamma} \quad w=g e t_{v^{\prime}}(\gamma)
$$

get $e, \varphi \Downarrow w, \gamma$
$\frac{\cdot \vdash e^{\prime}: \text { lens of }(\Sigma, R, P, F)}{. \vdash \operatorname{get} e^{\prime}:\{R\}}$
$\cdot \vdash v^{\prime}$ : lens of $(\Sigma, R, P, F)$
(1); T-GET
$\gamma: \Phi$
$w: \operatorname{Rel}(R, P, F)$
$\cdot \vdash w:\{R\}$
assumption
induction
induction
Lemma 65
def. $\operatorname{Rel}(R, P, F)$
case E-Put-Sat

```
\(e, \varphi \Downarrow v^{\prime}, \varphi_{1} \quad f, \varphi_{1} \Downarrow w, \varphi_{2} \quad \cdot \vdash v^{\prime}:\) lens of \((\Sigma, S, P, F)\)
\(w \models F \quad \forall r \in w . \operatorname{sat}(P, r) \quad \gamma=\) put \(_{v}\left(\varphi_{2}, w\right) \quad\) assumption
    put \(e\) with \(f, \varphi \Downarrow\) true, \(\gamma\)
\(\Gamma \vdash e:\) lens of \((\Sigma, R, P, F) \quad \Gamma \vdash f:\{R\}\)
    \(\Gamma \vdash\) put \(e\) with \(f\) :bool
\(\varphi_{1}: \Phi\)
    (3); induction (4)
\(\varphi_{2}: \Phi\)
\(\cdot \vdash w:\{R\}\)
\(w: \operatorname{Rel}(R, P, F)\)
def. \(\operatorname{Rel}(R, P, F)(6)\)
\(\cdot \vdash\) true:bool
\(\gamma: \Phi\)
case E-Put-Unsat
```

assumption
(1); T-Put
(3); induction (4)
(3) 4); induction (5)
(3. 4); induction
def. $\operatorname{Rel}(R, P, F)(6)$
(6); T-Const
(5); Lemma 66

```
case E-Put-Unsat
```

```
\(e, \varphi \Downarrow v, \varphi_{1}\)\begin{tabular}{c}
\(f, \varphi_{1} \Downarrow w, \gamma \quad v:\) lens of \((\Sigma, S, P, F)\) \\
\(w \not \models F\) or \(\exists r \in w . \neg \operatorname{sat}(P, r)\)
\end{tabular}
put \(e\) with \(f, \varphi \Downarrow\) false, \(\gamma\)
\(\Gamma \vdash e:\) lens of \((\Sigma, R, P, F) \quad \Gamma \vdash f:\{R\}\)
\(\Gamma \vdash\) put \(e\) with \(f\) : bool
\(\varphi_{1}: \Phi\)
(7); induction (8)
\(\gamma: \Phi\)
\(\cdot \vdash\) false: bool
\(e, \varphi \Downarrow v, \varphi_{1} \quad f, \varphi_{1} \Downarrow w, \gamma \quad v:\) lens of \((\Sigma, S, P, F)\)
    \(w \not \vDash F\) or \(\exists r \in w\). \(\neg \operatorname{sat}(P, r)\)
    put \(e\) with \(f, \varphi \Downarrow\) false, \(\gamma\)
    \(\vdash\) put \(e\) with \(f\) : bool
    \(1: \Phi\)
    (7. 8); induction
    T-Const
```

Full proof in Appendix D.1.

An additional requirement of the get and put functions is that they satisfy roundtripping guarantees. This property is specified in the following two lemmas.

Lemma 68 (GetPut). If $\varphi: \Phi$ and $\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$ then $\operatorname{put}_{v}\left(\varphi, \operatorname{get}_{v}(\varphi)\right)=$ $\varphi$.

Lemma 69 (PutGet). If $\varphi: \Phi$ and $\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$ and $M: \operatorname{Rel}(R, P, F)$ then $\operatorname{get}_{v}\left(\operatorname{put}_{v}(\varphi, M)\right)=M$.

Using the above properties we can show that get and put terms in our language also satisfy round-tripping. Care must be taken to ensure there are no sideeffecting computations that may alter the state of the database between the get
and put operations. We require that the lens is a value $v$, as the computation of a lens value may otherwise alter the database.

Theorem 13. Given a lens value $\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$ then we know that put $v$ with $(\boldsymbol{g e t} v), \varphi \Downarrow$ true,$\varphi$.

Proof.
$\cdot \vdash v:$ lens of $(\Sigma, R, F, A) \quad$ assumption
$v, \varphi \Downarrow v, \varphi$
E-Value
$M: \operatorname{Rel}(R, P, F)=\operatorname{get}_{v}(\varphi)$
Lemma 65
get $v, \varphi \Downarrow M, \varphi$ E-Get
$p^{2} t_{v}(\varphi, M)$
$=p u t_{v}\left(\varphi, \operatorname{get}_{v}(\varphi)\right)$
def. $M$
$=\varphi$
Lemma 68
put $v$ with $($ get $v), \varphi \Downarrow \operatorname{true}, \varphi$

Theorem 14. Given a lens value $\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$ and a view $M \in$ $\operatorname{Rel}(R, P, F)$ then we know that $(\lambda x$. get $v)(\boldsymbol{p u t} v$ with $M), \varphi \Downarrow M, \gamma$.

Proof.
$\cdot \vdash v:$ lens of $(\Sigma, R, F, A)$ assumption
$M: \operatorname{Rel}(R, P, F) \quad$ assumption
$v, \varphi \Downarrow v, \varphi$
E-Value
$\lambda x$. get $v, \varphi \Downarrow \lambda x$. get $v, \varphi$
E-Value
$\gamma=\operatorname{put}_{v}(\varphi, M)$
define
put $v$ with $M, \varphi \Downarrow$ true, $\gamma$
E-Put-Sat
get $v[$ true $/ x]=$ get $v$
def. subst
$\operatorname{get}_{v}(\gamma)=\operatorname{get}_{v}\left(\operatorname{put}_{v}(\varphi, M)\right)=M$
Lemma 69 (1)
get $v, \gamma \Downarrow M, \gamma$
$(\lambda x$. get $v)($ put $v$ with $M), \varphi \Downarrow M, \gamma$
(1); E-GET

E-App

```
Types \(\quad \tau::=\cdots \mid\) table of \((S, R)\)
Terms \(e, f::=\cdots \mid\) lens \(e\) of \(R\) with \(F\)
```

Figure 5.7: Links implementation.

It would also be possible to show that the language we specified is total using techniques such as by Altenkirch and Chapman [5], but practical languages such as Links are rarely total. The lenses themselves are essentially sum types, and Lemma 65 and 66 ensure that the get and put operations themselves are total if the arguments are.

### 5.6 Integration in Links

We implement relational lenses in Links. The Links implementation supports both incremental and naive relational lens semantics, and was used for the benchmarks in the evaluation presented in Section 3.3. Links can statically check the relational lens typing rules defined in this chapter. This section describes some of the specifics of the Links implementation.

Internally, Links relies on a version of relational lenses implemented in OCaml. Our OCaml implementation does not support any form of static type checking of the relational lens checks, but does include functions to perform the checks at runtime. The Links compiler extends the language with constructors for the various lenses, and uses the OCaml library to perform typechecking during compilation. When a Links program constructs a relational lens, the interpreter uses the OCaml library. Links supports executing code on both the web server and the client, but relational lenses are only supported in server side code.

Tables. Our relational lenses build on top of existing database support by reusing existing table handles. Links defines a primitive table expression which takes the relation name and a database handle, yielding a handle to a table in the database. The table expression assumes that the programmer has supplied a record type which corresponds to the types in the underlying database schema. Type table of $(S, R)$ is the type of a table handle with relation name $S$ containing records of type $R$. The programmer must take care not to construct lenses that
use multiple different databases, however this is not supported by existing query mechanisms either.

Lens Primitives. The lens primitive expression in Links requires the programmer to provide a table handle and specify the functional dependencies. The rule T-Lens is used to create a relational lens from a Links table. A lens primitive is assigned the default predicate constraint true. All columns referred to by a set of functional dependencies $F$, written names $(F)$, should be part of the table record type $R$.

$$
\begin{aligned}
& \text { T-LENS } \\
& \frac{\Gamma \vdash e: \text { table of }(S, R) \quad \bigcup \operatorname{names}(F) \subseteq \operatorname{dom}(R)}{\Gamma \vdash \text { lens } e \text { with } F: \text { lens of }(\{S\}, R, \text { true }, F)}
\end{aligned}
$$

Predicates. The programmer defines predicates as functions from records to boolean values. If the predicate does not depend on any unbound variables, then the record is known statically and all checks can be performed. If the record contains unbound variables then Links is unable to perform the predicate checks during compilation. In this case the programmer must explicitly acknowledge that the check is not performed during compilation by using the lenscheck constructor. This constructor may throw a runtime exception during application execution.

Evaluation The implementation was also used in a case study of the IUPHAR/BPS Guide to Pharmacology (GtoPdb) in Appendix E. The case study implements a curation interface for pharmacological targets in Links, making use of the relational lenses implementation described here. The curation interface is developed using the Model-View-Update (MVU) originally developed by Elm [1]. Relational lenses generate the data model used to populate the user interface, and the changed model from the user interface can directly be applied to the database. Development of the application is simplified by not requiring any special code to query the data or convert it either into or from a format required by the user interface.

### 5.7 Summary

The sequential style of relational lenses as defined by Bohannon et al. [12] is good for modelling transformations between whole database schemas, but is less optimal when working from a programming language. When using relational lenses in this setting, the programmer is more likely to work with a single view at a time. The programmer also does not provide the lens with a complete instantiated database, but instead expects the lens to query and update the database in a side-effecting manner.

As part of the language integration process, we propose treating relational lenses as handles to database views. Rather than sequentially composing lenses as in 2.2, our implementation uses sub-expressions to construct input lenses. The programmer is then able to use let-binding and lambda expressions to structure their code as desired. This approach also prevents the programmer from having to name intermediate views.

This chapter introduces language integrated lenses and provides typing rules. We provide a translation from our language integrated lenses to the sequential style lenses, and prove that a well-typed language integrated lens always constructs a well-typed sequential lens. The language integrated lenses also make use of the basic predicate syntax introduced in Chapter 4 , showing that concrete lenses and the corresponding checks satisfy all requirements of abstract lenses. We also show that our language preserves types during execution and satisfies the relational lens round-tripping guarantees.

The work presented in Chapter 4 and this chapter form the basis for both the Links implementation and the Haskell relational lens library. Chapter 6 describes the Haskell library in more detail.

## Chapter 6

## Relational Lenses as a Library

The work presented in this thesis is motivated by the task of turning relational lenses into a practical language feature. The two relational lenses implementations presented in this thesis are the only known implementations at the time of writing

The relational lens typing rules contain many small side conditions, making it easy for the programmer to forget to check some of them. These side-conditions can generally be checked mechanically though. Yet implementing the various checks required by relational lenses proves to be a challenging task.

A straightforward method to perform these checks would be to defer them to runtime, an approach proposed for predicates in Section 4.4. Doing so means that the program may still crash during execution if the lens is not correctly constructed. This places the burden of checking onto the programmer, who is required to thoroughly test the software with all possible inputs.

The implementation should therefore ideally perform compile-time checking of relational lenses. The implementation described in Section 5.6 is for the Links programming language [26]. However, executing the checks during compilation proves to be challenging. Simple type systems as provided by some of the most commonly used programming languages are not sufficient to implement something like the relational lenses checks. Bringing relational lenses to Links required additional changes to the core language. The extensions made to the Links compiler are specific to relational lenses, providing the flexibility required by the typing rules. This way of implementing relational lenses makes the first approach an ad-
hoc solution. Such domain specific compiler extensions increase the complexity of the core compiler, making it more difficult to maintain and show soundness of the compiler itself. Extending the compiler specifically for a single language feature is not desirable.

If relational lenses should be implemented without extending the compiler, it is necessary for the language to be extensible enough to support the static checks required by relational lenses. This chapter aims to demonstrate how relational lenses can be implemented as a library for Haskell.

The Glasgow Haskell Compiler (GHC) boasts a powerful type system supporting many useful features such as type-classes [87] and qualified types [53]. In addition, the type system supports many available extensions that can be used to equip the typechecker with additional capabilities. Some of these features which allow limited dependently typed programming help it to support a complex language feature such as relational lenses.

This chapter demonstrates how a combination of qualified types, type classes and type-families [77, 78] can be used to implement relational lenses as a library. The implementation requires no extension to the compiler despite its extensive compile-time checks which ensure that the relational lenses will be well-behaved.

There is an additional advantage of implementing relational lenses in such a more disciplined way. With the ad-hoc implementation it is not possible to provide types to functions where all components of a relational lens are known. Consider the following example:

$$
\begin{aligned}
& \text { from_date }:: \forall \Sigma R P Q F . \\
& \quad \text { int } \rightarrow \text { lens of }(\Sigma, R, P, F) \rightarrow \mathbf{l e n s} \text { of }(\Sigma, R, P \wedge Q, F) \\
& \text { from_date }=\lambda y l . \\
& \quad \text { select }_{\text {date }=y} \text { from } l
\end{aligned}
$$

The function from_date constructs a select lens, using the function argument $l$ as the underlying lens. This function is not well-typed because it is not possible to provide every lens to this function (e.g. what if the lens does not have a date column). This function is still useful though, because there are lenses it does accept. The function type is also not able to ensure that $Q$ is correctly instantiated to reflect the date $=y$ predicate introduced. Using qualified types, we can restrict
which lenses can be used by the function. With the additional restriction, the function is well-typed. The Haskell relational lenses allows the above function to be expressed as follows, where the Selectable (Var "date":= Erased '[] Int) s snew constraint ensures the function is only ever used correctly.

```
from_date ::
    Selectable (Var "date" := Erased '[] Int) s snew
    => Int -> Lens s -> Lens snew
from_date y l =
    select (var @"date" #= di y) l where
```

This chapter presents the end-to-end implementation of relational lenses in Haskell, which makes use of qualified types and also relies on the hybrid predicates presented in Section 4.4.1. The actual implementation also makes of incremental relational lens semantics for efficient database querying. The full source code can be found at https://github.com/rudihorn/haskell-lenses

An introduction to type-level programming in GHC is given in Section 6.1, including an example of how record types can be implemented in Section 6.1.5. Section 6.2 introduces various type-level data structures and functions used by relational lenses. Section 6.3 then provides an overview of how relational lenses can be used, and then describes how the constructors for relational lenses are implemented.

### 6.1 Type-Level Programming

For relational lenses to be well-behaved, certain conditions must apply to the constructed lens. These conditions, shown in the lens typing rules in Figure 2.7, are non-trivial and should be checked to ensure correct usage of the lens. While the program can perform the checks required during execution, the best time to perform the relational lens checks is during compilation of the program.

The relational lens side-conditions cannot be performed by standard unification type-checking, as the requirements go beyond simple type-checking. Instead, a form of general-purpose computation is required, so that algorithms such as tree cycle detection can be performed. This section looks at how GHC's type system can be used for tasks beyond the scope of traditional type systems.

### 6.1.1 Type level values

Haskell exposes modules for working with type-level literals1. Type-level literals allow the programmer to use values such as text and numbers within types. All type expressions have a kind, where kinds are to types as types are to values. Existing types that can be used to construct values are of kind $\star$. Text is of kind Symbol and numbers are of kind Nat.
type Hello = "Hello"
-- Hello has kind Symbol
type Five $=5$
-- Five has kind Nat
Existing data types can be promoted into a kind by prepending any constructor with a single quote ('). As the boolean type is defined as a variant data type, true or false of kind boolean can be used as types:

```
data Bool = False | True
```

-- Definition of Bool
bTrue = True
-- bTrue is a value true of type Bool

```
type BTrue = 'True
```

-- BTrue is the type true of kind Bool
Types must generally be attached to values to pass around. In some cases we may want to provide type arguments to functions that aren't attached to values. The Proxy type can be used as a type witness. It takes one type parameter $p$ of kind k and has a single constructor Proxy taking no values, making it equivalent to the unit value. The definition of the Proxy data type and a usage example constructing a value $v$ with the type argument 'True is shown below.

```
data Proxy (p :: k) = Proxy
```

v :: Proxy 'True = Proxy

[^3]
### 6.1.2 Type Families

Type families support type-level static computations [77, [78] . A type family is defined by pattern matching on type expressions. Recursion is supported, but partial applications and higher order functions require an additional extension [56]. Type families must be well kinded. The following is an example of a type family for calculating the length of a list and its usage. It makes use of the built-in + operator for performing number addition.

```
type family Length (l :: [k]) :: Nat where
    Length '[] = 0
    Length (_ ': xs) = 1 + Length xs
```

type Five = Length '[1, 2, 3, 4, 5]

This section also relies on the type level :++ infix operator, which performs a concatenation of two lists.

Another commonly used type family in the relational lenses implementation is a conditionally branching If type family from the Data.Type.Bool package, which is defined as follows:

```
type family If cond tru fls where
    If 'True tru fls = tru
    If 'False tru fls = fls
```


### 6.1.3 Type Classes

Type classes allow the generation of type-specific functions and values [87. They are used to generate functions which extract static type information and convert it into a runtime value. Type classes can also be used to generate functions which transform values with known static information, such as calculating the projection of a record which statically tracks the fields and types.

During various processing stages of relational lenses, as well as for other features such as printing values to the console, it is necessary to convert types into runtime values conveying the same information. The Recoverable i $t$ type class takes a type of kind $i$ and yields a value of type $t$.

The instance Recoverable (s :: Symbol) String is an example showing how to
convert Symbol types into String values using the Haskell built-in symbolVa\2. The function symbolval is provided by satisfying the KnownSymbol type class. The GHC compiler automatically provides the KnownSymbol class for all symbol constants, and similarly provides the KnownNat class for all type level natural numbers.

```
class Recoverable i t where
    recover :: Proxy i -> t
instance KnownSymbol s => Recoverable (s :: Symbol) String where
    recover p = symbolVal p
instance Recoverable 'True Bool where
    recover Proxy = True
instance Recoverable 'False Bool where
    recover Proxy = False
```

Values can be instantiated with explicit type signatures (val :: typ) or we can make use of explicit type applications using the @ symbol. The following are all invocations of the recover function with a type argument, but each using a different syntax:
bTrue = recover (Proxy :: Proxy 'True)
-- bTrue == True
bFalse = recover @'False Proxy
-- bFalse == False
sHello = recover @"Hello" Proxy
-- sHello == "Hello"

### 6.1.4 Constraints

Constraints are restrictions on types. Constraints can be used to prevent invalid type combinations and for deriving type classes.

Equality constraints $\tau_{1} \sim \tau_{2}$ specify that the types $\tau_{1}$ and $\tau_{2}$ should be equal.

[^4]Using the Length type family defined in Section 6.1.2, the function myfun requires a Proxy of a type level list containing exactly two elements.

```
myfun :: Length l ~ 2 => Proxy l -> ()
myfun Proxy = ()
```

We define a function which takes any type that is recoverable to a string by requiring the constraint Recoverable a String.

```
recover_string :: forall a. Recoverable a String => Proxy a -> String
recover_string Proxy = recover @a @String Proxy
```

Constraints can also be combined using tuple like notation, e.g. a function that recovers the string value of a list with exactly two elements can be defined as:

```
only_two :: forall l. (Length l ~ 2, Recoverable l [String])
    => Proxy l -> [String]
only_two Proxy = recover @l @[String] Proxy
```

The code can be simplified by defining a type alias for multiple constraints. For the above example we could define a type alias OnlyTwo which ensures that a type $l$ can be recovered into a string list and that the list only contains two elements.

```
type OnlyTwo l = (Length l ~ 2, Recoverable l [String])
only_two' :: forall l. OnlyTwo l => Proxy l -> [String]
only_two' Proxy = recover @l @[String] Proxy
```


### 6.1.5 Record Types

Relational lenses are used for processing tabular data. Tabular data contains collections of entries, called records, containing different values for the same set of columns. Recall the albums table from Section 1.1, which contains a collection of records with two columns, one called album containing a string and the other containing a number for the available quantity.

A record is a named and typed tuple of values. For each record the names and types of all the columns are known in the type, while the contents of individual columns are only known during runtime as part of the value. The record \{album: "Disintegration"; quantity: 6\} in the albums table has the type
\{album: String; quantity: Int\}. A table is a collection of records with the same type.

Existing work shows how extensible records can be implemented as a Haskell library [55], and various Haskell packages providing extensible records ${ }_{4}^{3} 4_{4}^{4}$ are available. The relational lenses library described here makes use of its own extensible records library, which is briefly described in this section. There is not necessarily anything novel about the implementation of extensible records here, but a custom implementation was used to ensure that all required operations are supported. This section is more helpful as an introduction to how type-level programming can be used in Haskell to implement features such as relational lenses.

Haskell supports a rich constraint based type system, allowing an approach comparable to work by Morris and McKinna [70] to be adopted. The idea is to describe the requirements on a row variable as a constraint on the row type rather than directly in the row type itself. This approach doesn't require extensions to the type unification system to support type inference such as in the approach by Rémy [76]. It also allows us to express more advanced row restrictions as constraints such as the relational join.

Instead of a projection function having the type signature \{album: String; -\} -> String the function might then instead have the type signature $\forall z_{1} .\{$ album : String $\} \triangleleft z_{1} . z_{1} \rightarrow$ String, denoting that the function expects a row of type $z_{1}$ such that \{album: String\} is a subtype of $z_{1}$. However, unlike the approach by Morris and McKinna [70], our constraints are more explicit for operations the row should support. An example would be the type constraint Project $r$ "album" String, denoting it should be possible to project $r$ on the column album which should return a String.

The first definition required is a kind for storing the column information called Env. The columns are stored as a list of tuples, where each column tuple contains the name, and type of kind $\star$. Changes to the environment can be performed by defining type families on Env kinds. The definition of the Env kind is shown in Figure 6.1.

[^5]```
type Env = [(Symbol, *)]
data Row (e :: Env) where
    Empty :: Row '[]
    Cons :: (Ord t, Eq t) => t -> Row env -> Row ('( key, t) ': env)
```

Figure 6.1: The definition of the Env kind and the Row type.

A row value is constructed similarly to lists; any instance is either an empty row which has an empty environment, or a cons constructor with a value of type $t$ extending an existing row, whose environment is extended by a new tuple with a given name and the data type t . The two constructors ensure that the corresponding values are well typed with respect to Env. More complicated operations such as projections can then be defined using type classes.

The following code snippet demonstrates the usage of records. A record can be constructed using the Cons and Empty constructors, as shown in the first example. A user friendly syntax, shown in the second example, allows the record to be constructed from a tuple using the toRow function. The user-friendly syntax has type class instances for various tuple sizes. Both examples are used to construct the record $\{A=5, B=" h "\}$. The labels for the field names are only provided in the type information.

Records contain an implementation for the Show type class to support prettyprinting into a form $\{$ field $=$ value, $\ldots\}$.

```
-- Examples
row1 :: Row '[ '( "A", Int), '("B", String)]
row1 = Cons 5 (Cons "h" Empty)
-- equivalent friendly syntax
type MyRow = @'[ '( "A", Int), '("B", String)]
row1' = toRow @MyRow (5, "h")
-- pretty printing
row1
> { A = 5, B = "h" }
```

```
-- projection
fetch @"A" row1
> 5
update @"C" 20 row1
> {C = 20, A = 5, B = "h" }
```

The above snippet also shows an example of how a record can be projected onto a field. This is done using the fetch function, whose type signature is shown in the snippet below. This function requires the constraint Fetchable $s$ env $t$ evid to be satisfied, which ensures that there is a field $s$ of type $t$ given a record of type env. Details of how the fetch function works are shown later in this Section. The other example shows how a record can be updated. In this example field $C$ is set to the value 20 on record row 1 . The function signature of the update function is also shown in the snippet below.

```
fetch :: forall s env t evid. Fetchable s env t evid => Row env -> t
update :: forall s t env tnew evid. Updatable s t env tnew evid =>
    t -> Row env -> Row tnew
```

Fetch The fetch function should return the value of a column specified by the programmer. The desired column is provided as a type-level argument. In order to define a type class that projects out the column, a representation of the location of the column is required. The InEnvEvid data type defines a path to the requested field. The data type has two constructors. A Take constructor specifies that the head (first value attached to a Cons constructor) should be returned. Alternatively a Skip evid value indicates that the first cons element should be ignored, and evid is a path to the required column in the remaining elements. This datatype is equivalent to the natural numbers, and it would be possible to use the builtin Nat instead, but we use a special data type for clarity.

```
data InEnvEvid where
```

    Take : : InEnvEvid
    Skip :: InEnvEvid -> InEnvEvid
    To fetch the column of a record, it is necessary to construct the path to that column first. The Find env stype family looks up the field $s$ in the row type env. It returns a path of type InEnvEvid to that field.

```
type family Find (env :: Env) (s :: Symbol) :: InEnvEvid where
    Find ('(key, _) ': env) key = 'Take
    Find (_ ': env) key = 'Skip (Find env key)
```

The evidence path is used to determine the type of the field in the record. The type family EvidType env s returns the field type of kind $\star$ for a given record env and an evidence path $s$. The type family just follows the path and returns the second component of the corresponding tuple. It is used t

```
type family EvidType (env :: Env) (s :: InEnvEvid) :: * where
    EvidType ('(_, val) ': _) 'Take = val
    EvidType (_ ': xs) ('Skip evid) = EvidType xs evid
```

The next step is to implement a type class FetchRow $t$ i $r$ that returns the record field value of type $t$ given an evidence path $i$ for a given record value $r$. It defines two instances, one for a 'Take value which matches on a Cons $v$ _ and returns the current value $v$. The other instance matches on 'Skip evid and calls intfetch using the evidence evid on the tail of the row.

```
class FetchRow t (i :: InEnvEvid) r where
    intfetch :: r -> t
instance FetchRow t 'Take (Row ('(s, t) ': env)) where
    intfetch (Cons v _) = v
instance (FetchRow t evid (Row env)) =>
    FetchRow t ('Skip evid) (Row ('(so, to) ': env)) where
    intfetch (Cons _ row) = intfetch @t @evid row
```

The final step is to wrap up everything in a single constraint alias Fetchable $s$ env $t$ evid as well as a function fetch using the constraint. The constraint first looks up an evidence path evid using the Find type family. An equality constraint ensures that the type $t$ is equal to the type looked up by the EvidType type family, and there must also be an instance of the FetchRow type class for the given record and type. The fetch function can then just use the intfetch function from the FetchRow type class to retrieve the value.

```
type Fetchable s env t evid = (
    evid ~ Find env s,
```

```
t ~ EvidType env evid,
FetchRow t evid (Row env))
```

```
fetch :: forall s env t evid. Fetchable s env t evid => Row env -> t
fetch row = intfetch @t @evid row
```


### 6.2 Relational Lens Kinds

The relational lens typing rules require conditions on the lens invariants and inputs to be met. This includes restrictions on the row type of the view, functional dependencies, referenced tables as well as a predicate restriction on the view. For these checks to be performed during compilation, a type-level representation of these invariants is required. This section introduces the data types required, as well as some common checks required on them.

### 6.2.1 Tables

Relational lenses need to ensure that no underlying tables are used twice in a single relational lens expression. Relational lenses keep track of the table names used by a lens expression. Each table name is a string value, we define the Table kind as a Symbol to represent this.

```
type Table = Symbol
```

A collection of tables is represented using a list of tables. The kind tables is defined as [Table].

```
type Tables = [Table]
```

The IsDisjoint type family is used to show that the two collections of tables are disjoint. The type family returns a boolean value indicating if the two collections of symbols are disjoint. This constraint is defined as a predicate called DisjointTables.

```
type family IsElement (s :: k) (r :: [k]) :: Bool where
    IsElement _ '[] = 'False
    IsElement x (x ': xs) = 'True
    IsElement x (y ': ys) = IsElement x ys
```

```
type family IsDisjoint (l :: [Symbol]) (r :: [Symbol]) :: Bool where
    IsDisjoint '[] _ = 'True
    IsDisjoint (x ': xs) ys = Not (IsElement x ys) && IsDisjoint xs ys
    type DisjointTables ts1 ts2 =
        IsDisjoint ts1 ts2 ~ True
```


### 6.2.2 Functional Dependencies

Functional dependency constraints $X \rightarrow Y$ are restrictions on tabular data specifying that all records with the same values for the fields $X$ should have the same values for the fields $Y$. Functional dependencies are all tracked statically in the type system. The following data type is used to represent functional dependencies.

```
data FunDep where
    FunDep :: [Symbol] -> [Symbol] -> FunDep
```

The infix operator --> is used to construct functional dependencies by the programmer. It is defined as user friendly syntax rather than using the constructor FunDep. The friendly syntax is also used to normalize the functional dependencies, so that all duplicate are removed and the columns are stored in alphabetical order using the SymAsSet type family.

```
type family (-->) (left :: [Symbol]) (right :: [Symbol]) :: FunDep where
    xs --> ys = ('FunDep (SymAsSet xs) (SymAsSet ys) :: FunDep)
```

The type families Left and Right are used to project a functional dependency $X \rightarrow Y$ onto its left component $X$ or right component $Y$. The type families Lefts and Rights do the same for lists of functional dependencies. Finally, given a collection of functional dependencies, all referenced columns can be determined using the Cols type family.

```
type family Left (f :: FunDep) :: [Symbol] where
Left ('FunDep left _) = left
type family Right (f :: FunDep) :: [Symbol] where
Right ('FunDep _ right) = right
```

```
    type family Rights (f :: [FunDep]) :: [[Symbol]] where
    Rights '[] = '[]
    Rights (x ': xs) = Right x ': Rights xs
    type family Lefts (f :: [FunDep]) :: [[Symbol]] where
    Lefts '[] = '[]
    Lefts (x ': xs) = Left x ': Lefts xs
type family Cols (f :: [FunDep]) :: [Symbol] where
    Cols '[] = '[]
    Cols ('FunDep l r ': fds) = l :++ r :++ Cols fds
```

In most cases we work with collections of functional dependencies in the form [FunDep]. An important operation on functional dependencies is the transitive closure of a collection of symbols. The following code snippet shows how the type family TransClosure st fds is defined, which calculates the transitive closure starting with a set of columns st for the functional dependencies $f d s$. The transitive closure type family uses the Closure type family which non-transitively adds the right side of each functional dependency whose left side is a subset of the given columns. The closure type family is applied $n$ times by the TransClosureF family. By using the count of functional dependencies for $n$, it is ensured that the full transitive set is calculated.

```
type family Closure (from :: [Symbol]) (to :: [FunDep]) :: [Symbol] where
    Closure fr '[] = fr
    Closure fr (f ': fds) = If (IsSubset (Left f) fr)
        (SymUnion (Right f) (Closure fr fds))
        (Closure fr fds)
type family TransClosure (st :: [Symbol]) (fds :: [FunDep]) :: [Symbol] where
    TransClosure fr fds = TransClosureF fr fds (Len fds)
type family TransClosureF (st :: [Symbol]) (fds :: [FunDep]) n :: [Symbol] where
    TransClosureF fr fds 0 = fr
    TransClosureF fr fds n = (TransClosureF (Closure fr fds) fds (n-1))
```

Another important type family on functional dependencies is the Outputs functional dependency, which returns all the columns which may be restricted by a functional dependency. This function is implemented following the definition from Figure 2.4

```
type family OutputsL (fds :: [FunDep]) where
    OutputsL '[] = '[]
    OutputsL (fd ': fds) = (SetSubtract (Right fd) (Left fd)) :++ OutputsL fds
```

type family Outputs (fds :: [FunDep]) where
Outputs fds = SymAsSet (OutputsL fds)

Many lenses require the tree form check on functional dependencies. This check requires the functional dependencies to form a forest of nodes, where all nodes are disjoint from each other. This check is performed by ensuring that for all functional dependencies $X \rightarrow Y$, the right component $Y$ 's are disjoint from each other, as well as the set (i.e. with all duplicates removed) of all $X$ 's and $Y$ 's are also disjoint from each other. Finally the set of functional dependencies should not have any cycles, which is checked using the IsAcyclic type family. The definitions of AllDisjoint and IsAcyclic are available in the source code, as they are long.

```
type family IsInTreeForm (fds :: [FunDep]) where
    IsInTreeForm fds =
        AllDisjoint (Rights fds) &&
        AllDisjoint (SLAsSet (Rights fds :++ Lefts fds)) &&
        IsAcyclic fds
```


### 6.2.3 Predicates

The select lens allows the user to filter out records using predicates. Predicates are functions from records to booleans, indicating if the record should be included (true) or filtered out (false). Any predicate should be equivalent to testing for the membership of the record in a (possibly infinite) fixed relation, restricting which predicates can be expressed. For example, predicates cannot depend on any other records or unfixed values, preventing the use of aggregates or the dependency on other relational tables.

In the Links implementation of relational lenses, predicates are defined as functions in the regular language syntax [51]. The predicates can be defined as regular functions, which are then normalised into a lens predicate as presented in Section 4.3. Recall the basic predicate language from Section 4.1 in Figure 6.2. Predicates can either be a constant value $c$, a field label $\ell$, an n-ary operator $\odot$ applied to its arguments, or an if-then-else statement.

$$
P, Q::=c|x \cdot \ell| \odot\{P\} \mid \text { if } P_{1} \text { then } P_{2} \text { else } P_{3}
$$

Figure 6.2: The basic predicate predicate language introduced in Chapter 4 .

The Haskell library implementation of relational lenses is presented using the hybrid predicates technique introduced in Section 4.4.1. There are some differences to the version presented in Section 4.4.1. Rather than using a form of quasi-quotation which would likely require something like Template Haskell, this implementation provides distinct constructors for static, dynamic and hybrid predicate expressions. It is also necessary to differentiate between static and dynamic values due to technical reasons. Both static and dynamic predicates use the same underlying data structure, and as these data types are considered internal, it is ensured that no erased dynamic predicates will be constructed. Finally, the hybrid predicates described in Section 6.2.3 are actually just construction functions.

## Predicate Values

Predicates require a value kind to store constant values and to perform evaluation. Value kinds are used to box values. The implementation requires two datatypes. The first datatype DValue stores a dynamic value that is only known during runtime. In dynamic predicates this means having a single type to store any possible primitive value, such as a string or an integer. DValue has a constructor for boxing each value type. Each constructor takes a corresponding runtime value and returns a DValue.

```
data DValue where
    Bool :: Bool -> DValue
    Int :: Int -> DValue
    String :: String -> DValue
```

The SValue datatype is used to contain compile-time predicate values. It is required because the String and Symbol types/kinds differ for runtime and compile time values. Similarly, the Nat and Integer also differ. The new datatype also contains a constructor for each static primitive type.

```
data SValue where
    Bool :: Bool -> SValue
    Int :: Nat -> SValue
    String :: Symbol -> SValue
```


## Predicate Kind

Predicates are implemented using the abstract data type defined in Figure 6.3. Unlike the work in Section 4.4.1, the same underlying datatype is used for both static and dynamic predicates, but with type variables to differentiate between static and dynamic value types. The data type depends on two type variables, id for the identifier type and $v$ for the value type. Static identifiers are represented using the Symbol type, while runtime identifiers are represented using the String type. Similarly the value type variable is defined as SValue and DValue respectively. Constants take a value representing the constant, variables take an identifier, unary and infix operator applications take the operator and one or two arguments respectively. A case statement is a more general, but equivalent, constructor for if statements. The Erased constructor is used for hybrid predicates and is explained further in Section 6.2.3.

The type DPhrase representing dynamic predicates is defined as a Phrase with identifiers of type String and values of type DValue. SPhrase on the other hand represents static phrases, where identifiers are of kind Symbol and values are of kind SValue. Predicates are converted into values for querying by implementing the type class Recoverable SPhrase DPhrase.

For lenses with predicates it is necessary to ensure that the predicates are well typed. The type family Typ is defined, which takes a typing environment (equivalent to a row type) as well as a predicate and returns either some type if the environment is well typed, or nothing if the predicate is ill-typed. In practice this requires validating that for a given lens' row type rt and predicate $p$ the constraint (Typ rt p ~ 'Just 'Bool) holds. We omit the definition of Typ, as it is

```
data Phrase id v where
    Constant :: v -> Phrase id v
    Var :: id -> Phrase id v
    UnaryAppl :: UnaryOperator -> Phrase id v -> Phrase id v
    InfixAppl :: Operator -> Phrase id v -> Phrase id v -> Phrase id v
    Case :: Maybe (Phrase id v) -> [(Phrase id v, Phrase id v)]
                            -> Phrase id v -> Phrase id v
    Erased :: Env -> * -> Phrase id v
```

type SPhrase = Phrase Symbol SValue
type DPhrase = Phrase String DValue

Figure 6.3: The data type defining static and dynamic predicates.
straightforward and provided in the code.
type family Typ (env :: Env) (phrase :: SPhrase) :: Maybe Type
Evaluation is also defined for static predicates. The type family Eval takes an evaluation environment, which is a collection of label and value pairs, as well as a static phrase and returns a value if the expression is static and well typed. The definition is fairly straightforward and is available in source code.
type family Eval (env :: EvalEnv) (phrase :: SPhrase) :: Maybe Value
In addition a type family to determine the set of free type variables is required. The FTV type family takes as static phrase and returns a list of symbols that can be found in the predicate. The definition of FTV is a straightforward traversal of the syntax tree and has also been left out.
type family FTV (phrase :: SPhrase) :: [Symbol]

## Drop Lens checks

The drop lens requires more in-depth checks on lens predicate constraints, which are covered in Section 4.2. For a drop lens to behave correctly, it requires evidence that the equivalent relation of the predicate for the underlying lens forms a lossless join decomposition of the dropped column and all remaining columns. In predicate terms, a sufficient but not necessary condition is to check that for any predicate it is either a logical conjunction $P \wedge Q$ and both $P$ and $Q$ are loss-
less join decompositions, or the predicate can be typed under either the dropped column or all columns except the dropped column.

The IsLJDI type family is used to check the lossless join decomposition property on static predicates. IsLJDI matches on two cases:

1. If the predicate is $P \wedge Q$ then both $P$ and $Q$ must be a lossless join decomposition.
2. For any other predicate, the free variables must either be a subset of the dropped columns or completely disjoint from them.

The second condition is a slight variation from the definition in Section 4.2.2, as only the set of dropped columns are provided. If the set of freely bound variables is not a subset of the dropped columns, it must be completely disjoint, relying on the fact that $R_{1}$ and $R_{2}$ must be disjoint in the other definition. The lossless join decomposition check is formally defined as follows.

```
type family IsLJDI (vs :: [Symbol]) (phrase :: SPhrase) :: Bool where
    IsLJDI vs ('InfixAppl 'LogicalAnd p1 p2) =
        IsLJDI vs p1 && IsLJDI vs p2
    IsLJDI vs p = IsSubset (FTV p) vs || IsDisjoint (FTV p) vs
```

type family LJDI (vs :: [Symbol]) (phrase :: SPhrase) where
LJDI vs $\mathrm{p}=$ IsLJDI vs p ~ 'True

The drop lens also requires that the supplied default values for the dropped columns satisfy the predicate constraints. The used definition for IsLJDI simplifies the check from Section 4.2 to the following three cases:

1. If the predicate is $P \wedge Q$ then perform the default value check on $P$ and $Q$.
2. If the predicate only depends on the dropped columns, then evaluating it on the default value should return true.
3. If the predicate does not depend on the dropped columns, then it no further checks are necessary.

Formally we define the default value check IsDefVI as follows. The UnpackTrue type family returns 'True if its argument is equal to 'Just ('Bool 'True) or 'False otherwise.

```
type family IsDefVIEx (subs :: Bool) (disj :: Bool) (env :: EvalEnv)
    (phrase :: SPhrase) :: Bool where
    IsDefVIEx 'True _ env p = UnpackTrue (Eval env p)
    IsDefVIEx 'False 'True env p = 'True
    IsDefVIEx _ _ _ _ = 'False
type family IsDefVI (env :: EvalEnv) (phrase :: SPhrase) :: Bool where
    IsDefVI env ('InfixAppl 'LogicalAnd p1 p2) =
        IsDefVI env p1 && IsDefVI env p2
    IsDefVI env p = IsDefVIEx (IsSubset (FTV p) (Vars env))
        (IsDisjoint (FTV p) (Vars env)) env p
type family DefVI (vs :: [Symbol]) (phrase :: SPhrase) where
    DefVI vs p = IsDefVI vs p ~ 'True
```

Finally, a type family is required to remove all terms in a static predicate that refer to the removed columns. The type family ReplacePredicate traverses all conjunction terms in a predicate, and if the term only depends on the set of dropped columns then the term in the conjunction is replaced with the default term true.

```
type family ReplacePredicate (env :: EvalEnv) (phrase :: SPhrase) :: SPhrase where
    ReplacePredicate env ('InfixAppl 'LogicalAnd p1 p2) =
        ReplacePredicate env p1 :& ReplacePredicate env p2
    ReplacePredicate env p = If (IsSubset (FTV p) (Vars env)) (B 'True) p
```


## Hybrid predicates

Hybrid predicates have an internal (i.e. not allowed to be used by the programmer) constructor HPred which takes a dynamic predicate dp and returns a hybrid predicate of type HPhrase $p$, where $p$ is an arbitrary static predicate.

```
data HPhrase (p :: SPhrase) where
    HPred :: DPhrase -> HPhrase p
```

The public programming interface for hybrid predicates contains the same set of constructors for static predicates and applies the same operation to both the static and the dynamic predicate. The following is an example of the (\#>) infix
operator for the greater than comparison. When a greater than hybrid predicate is constructed, it returns a new hybrid predicate the dynamic predicate is the combination of both underlying dynamic predicates combined with a greater than infix operator application. The static predicate is also generated by combining the two static predicates with a greater than binary operator $(:>)$.

```
(#>) :: forall p1 p2. HPhrase p1 -> HPhrase p2 -> HPhrase (p1 :> p2)
(HPred p1) !> (HPred p2) = HPred $ P.InfixAppl P.GreaterThan p1 p2
```

The concrete information of a hybrid predicate can be erased so that only the type information of the predicate is retained. The additional predicate constructor Erased env ret, which is only considered valid for static predicates, indicates a predicate with an unknown body, but which is known to type to the return type ret given an environment env.

```
data Phrase id v where
```

```
    Erased :: RT.Env -> T.Type -> Phrase id v
```

The programmer can erase a predicate using the erase function, which takes a hybrid predicate and removes the function body in the type. The static predicate is replaced with an Erased constructor that only contains the typing information of the predicate. The erase function ensures that the predicate is well-typed using the Typ constraint. The underlying dynamic predicate is left unchanged.

```
erase :: forall rt ret (p :: SPhrase). (P.Typ rt p ~ 'Just ret)
    => HPhrase p -> HPhrase ('P.Erased rt ret)
erase (HPred p) = HPred p
```

In the hybrid predicates formulation in Section 4.4.1, an anti-quotation operator is used to insert runtime values into the predicates as literals. Here, specific constructors to perform this task are introduced instead. The following example shows how an integer literal predicate can be constructed from an Int value provided at runtime. The function constructs a dynamic underlying predicate with the integer value, while the static predicate is constructed as an Erased predicate that returns an integer value.
di :: Int -> HPhrase ('P.Erased '[] 'T.Int)
di $\mathrm{v}=$ HPred (P.Constant \$ DP.Int v)

The FTV function can be extended to support the Erased constructor. This allows the DefVI and LJDI type families to still be used on predicates that contain erased parts.

The following snippet shows an example of how a function can be defined that constructs a predicate depending on the input parameters $b, i$ and $s$. If $b$ is true, then the predicate is chosen to be quantity $>i$, where $i$ refers to the function parameter. Otherwise the predicate is chosen to be $a l b u m=s$, where $s$ refers to the function parameter. These predicates can be unified to the result of the if statement, because they are both erased to predicates taking a quantity and album field, returning a boolean value. The predicate still satisfies the lossless join decomposition requirement when constructing the drop lens, because the predicate does not refer to the dropped column date.

```
type PredRow = '[ '("quantity", 'T.Int), '("album", 'T.String)]
my_lens b i s = do
    testdb tracks3 where
    pred = if b
            then (erase @PredRow @'T.Bool (var @"quantity" #> di i))
            else (erase @PredRow @'T.Bool (var @"album" #= ds s))
    tracksl = join tracks albums
    tracks2 = select pred tracks1
    tracks3 = dropl @'[ '("date", 'P.Int 2020)] @'["track"] tracks2
```

Currently the user is required to explicitly specify the erased typing information. By extending the type system to support basic polymorphism it could be possible to remove the requirement of having explicitly declared typing environments. This would likely require a type-checking plugin extension to allow the unification of erased static predicates.

## Other Checks

Another requirement used by multiple lenses ensures that information propagating via functional dependencies doesn't cause records to violate the predicate constraint. This property can be checked by ensuring that the predicate does not depend on any columns that are considered outputs of a functional dependency.

In practice, this check ensures that the set of columns referred to by the predicate is disjoint from the output columns of the functional dependencies calculated using the Outputs type family.

```
type family IgnoresOutputs (phrase :: SPhrase) (fds :: [FunDep]) :: Bool where
```

    IgnoresOutputs p fds = IsDisjoint (FTV p) (Outputs fds) ~ 'True
    
### 6.2.4 Lens Sort

The lens sort is a kind describing the compile-time information required by a lens to perform typechecking. The sort is a tuple of the used tables, the view's record type, a static predicate restriction and a collection of functional dependencies. It only has a single constructor Sort taking each component as an argument.

We define a type family to project the sort onto each of its components. Ts sort gets the tables used by the lens, Rt sort returns the row type, P sort returns the predicate restricting the view and Fds sort returns the functional dependency constraints. We show the sort kind the projection type family Ts sort below. The remaining projections functions Rt, P and Fds can be defined similarly to Ts.

```
data Sort where
    Sort :: Tables -> Env -> SPhrase -> [FunDep] -> Sort
-- get tables
type family Ts (s :: Sort) :: Tables where
    Ts ('Sort ts _ _ _) = ts
```


### 6.3 Using Relational Lenses

Before detailing how relational lenses work, this section presents an intuition on how relational lenses can be used from a programmer perspective.

### 6.3.1 Syntax

Figure 6.4 presents the syntax used by relational lenses.
The presented syntax is not used for the implementation, and is only provided as an overview of how relational lenses can be defined, combined and used. For all
expressions it is possible to abstract types by using type aliases and to abstract expressions using let binding or function abstractions.

The <lens> token is used to construct lenses. Lens expressions can either construct a primitive lens, a select lens, a join lens or a drop lens.

The primitive lens requires the name of the table it represents. This is provided as a string type-level argument. The programmer must also provide the row type using a <row> token, which expects a list of key and type tuples the lens expects. Finally the functional dependencies are provided as an $\langle f d s\rangle$ token, which is a list of functional dependencies.

The select lens expects a predicate expression which can be constructed using the $<$ pred $>$ token. The predicate expressions are differentiated from other identical expressions using a \# symbol.

The join lens expects two further lens expressions. Lastly the drop lens requires a type-level record $\langle$ env> token with the columns to be dropped, where the value of the record indicates the default value. It also requires the list of columns that define the dropped columns to be provided as a list of symbols $\langle i d s\rangle$.

Lenses can actually be used with <query> tokens. The get operation queries the lens using a given connection <conn>, while the put operation submits the updated view in the form of a <data> token to the given lens using the connection provided. Syntax for the <conn> token is not provided, as this depends on the database provider used. The <data> token is also not specified, as this could be any code that constructs or alters a view.

### 6.3.2 Database Connection

The presented relational lenses library relies on the postgresql-simple ${ }^{5}$ library for performing database interactions. Performing get and put require a Connection value (<conn> in Figure 6.4). As the connection is established using the third party library, we only give an example of how to connect to the database. The following code yields a Connection in the IO monad [86].
connect defaultConnectInfo \{
connectDatabase = <database>,

[^6]```
<table> ::= <string>
<id> ::= <string>
<typ> ::= Int | Bool | String
<col> ::= '(<id> , <typ> )
<row> ::= '[] | '[ <col> ( , <col> )* ]
<ids> ::= '[] | '[<row> @<typ> <pred>@*) <id> (, <id> )* ]
<fd> ::= <ids> --> <ids>
<fds> ::= '[] | '[ <fd> ( , <fd> )* ]
<val> ::= 'P.Int <int> | 'P.String <string> | 'P.Bool <bool>
<asign> ::= '( <id> , <val> )
<env> ::= '[] | '[ <asign> (, <asign> )* ]
<pred> ::= <pred> #= <pred> | <pred> #> <pred> | <pred> #< <pred>
    | <pred> #& <pred> | <pred> #| <pred> | <pred> #+ <pred>
    | #<label> | neg <pred> | i @'<int> | s @'<string> | b @'<bool>
    | di <int expr> | ds <string expr> | db <bool expr>
    | erase @<row> @<typ> <pred>
<lens> ::= prim @<table> @<row> @<fds>
    | select <pred> <lens>
    | join <lens> <lens>
    | dropl @<env> @<ids> <lens>
<query> ::= get <conn> <lens> | put <conn> <lens> <data>
```

Figure 6.4: Syntax of relational lenses in Haskell.

```
connectUser = <user>,
connectPassword = <password>
}
```


### 6.3.3 Lens Constructors

This section presents type signatures and restrictions of each lens constructor. Existing work on language integrated relational lenses [51] explains in more detail why these restrictions are required.

Primitive Lens The primitive lens is used to declare an existing table within the database. The user supplies a static string referring to the table name within the database, a row type and a set of functional dependencies.

Many of the type constructors contain arguments which do not need to be specified by the user. These are greyed out.

```
prim :: forall table rt fds s.
    (s ~ 'Sort '[table] rt DefaultPredicate fds,
    Lensable s)
    => Lens s
```

The primitive lens typing rule first constructs a lens sort using the provided functional dependencies and row type. The primitive lens constructor uses the default predicate accepting all records. An additional constructor can be provided that accepts a predicate, which is useful if the database contains other constraints on the table contents. The Lensable constraint should perform all checks required by the primitive Lens typing rule from Section 5 .

$$
\begin{aligned}
& \text { T-LENS } \\
& \frac{\Gamma \vdash e: \text { table of }(S, R) \quad \bigcup \operatorname{names}(F) \subseteq \operatorname{dom}(R)}{\Gamma \vdash \text { lens } e \text { with } F: \text { lens of }(\{S\}, R, \text { true }, F)}
\end{aligned}
$$

Primitive lenses only require basic sanity checks. The sanity checks ensure that the record type does not contain any duplicate columns and that the functional dependencies of the lens only refer to existing columns. If an arbitrary predicate is provided, the constraint ensures that it is well-typed. The LensableImplConstraints alias ensures that all type class constraints that are required for technical reasons have instances, but the definition of it is left out.

```
type Lensable s =
    (P.Typ (Rt s) (P s) ~ 'Just Bool,
    Subset (Cols (Fds s)) (VarsEnv (Rt s)),
    LensableImplConstraints s)
```

Join Lens The join lens takes two lenses and produces a lens performing the natural join between the two underlying views. The difficulty with view-updates on join lenses is that defining the put semantics are ambiguous. Bohannon et al. [12] define multiple variants, and we use the delete left variant, but any deletion semantics can be used. The delete left variant resolves ambiguous record deletion behaviour by only deleting from the left table.

The join lens has the type signature shown in the snippet below. The programmer is not required to specify any type arguments explicitly. The join function takes two lenses with lens sorts s1 and s2 and returns a lens with the sort snew, constrained by the Joinable constraint.

```
join :: Joinable s1 s2 snew joincols =>
    Lens s1 -> Lens s2 -> Lens snew
```

To understand the Joinable constraint, first recall the typing rule for the join lens from Section 5 .

T-Join-Left

| $\Gamma \vdash e:$ lens of $(\Sigma, R, P, F)$ | $\Gamma \vdash f:$ lens of $\left(\Delta, R^{\prime}, Q, G\right)$ |
| :---: | :---: |
| $G \vDash \operatorname{dom}(R) \cap \operatorname{dom}\left(R^{\prime}\right) \rightarrow \operatorname{dom}\left(R^{\prime}\right)$ | $F$ is in tree form $\quad G$ is in tree form |
| $P$ ignores outputs $(F) \quad Q$ ignores outputs $(G)$ | $\Sigma \cap \Delta=\varnothing$ |
| $\Gamma \vdash$ join_dl $e$ with $f:$ lens of $\left(\Sigma \cup \Delta, R \cup R^{\prime}, P \wedge Q, F \cup G\right)$ |  |

The required constraints for the join lens are shown in Figure 6.5. The Joinable constraint expand out the lens sorts and applies them to the JoinableExp constraint. It is also used to construct the new sort type based on the old sorts. The following conditions are checked:

The disjointedness of tables check $\Sigma \cap \Delta=\varnothing$ is performed using the DisjointTables ts1 ts2 constraint.

The overlapping columns of two tables are the join columns, and they should have identical types for both, which is ensured by the OverlappingJoin rt1 rt2 constraint.

```
type JoinableExp ts1 rt1 p1 fds1 ts2 rt2 p2 fds2 rtnew joincols =
    (rtnew ~ JoinEnv rt1 rt2,
    joincols ~ R.InterCols rt1 rt2,
    DisjointTables ts1 ts2,
    OverlappingJoin rt1 rt2,
    IgnoresOutputs p1 fds1, IgnoresOutputs p2 fds2,
    Subset (VarsEnv rt2) (TransClosure joincols fds2),
    InTreeForm fds1, InTreeForm fds2,
    JoinImplConstraints ts1 rt1 p1 fds1 ts2 rt2 p2 fds2 rtnew joincols)
type Joinable s1 s2 snew joincols =
    (JoinableExp (Ts s1) (Rt s1) (P s1) (Fds sl)
                        (Ts s2) (Rt s2) (P s2) (Fds s2)
                        (Rt snew) joincols,
    snew ~ 'Sort (Ts s1 :++ Ts s2) (JoinEnv (Rt s1) (Rt s2))
            (Simplify (P s1 :& P s2)) (SplitFDs (Fds s1 :++ Fds s2)))
```

Figure 6.5: The constraints required by the join lens.

The check $G \vDash \operatorname{dom}(R) \cap \operatorname{dom}\left(R^{\prime}\right) \rightarrow \operatorname{dom}\left(R^{\prime}\right)$ ensures that the functional dependencies transitively define the right table by the join key. The constraint Subset (VarsEnv rt2) (TransClosure joincols fds2) checks this condition, where the type family VarsEnv returns a list of symbols given a row type.

The tree form checks on functional dependencies are ensured using the InTreeForm type families.

The ignores outputs checks on predicates are performed using the IgnoresOutputs type family.

All type class instances that need to be checked for implementation purposes are summarized in the JoinImplConstraints constraint. As these are mostly technical requirements they are not covered in detail here.

Select Lens To construct a select lens, the programmer provides a hybrid predicate that evaluates under the context of a record in the table to a boolean value. If the value is true then the record is included in the output.

```
select :: forall p s snew.
    (Selectable p s snew) => HPhrase p -> Lens s -> Lens snew
    select pred \(\mathrm{l}=\) Select pred l
T-Select
    \(\Gamma \vdash e:\) lens of \((\Sigma, R, P, F) \quad R \vdash Q\) : bool
    \(F\) is in tree form \(\quad P\) ignores outputs \((F)\)
    \(\Gamma \vdash \operatorname{select}_{Q}\) from \(e:\) lens of \((\Sigma, R, P \wedge Q, F)\)
```

The Selectable p s snew constraint takes the static predicate $p$ of a hybrid predicate, as well as the sort of the underlying lens $s$ and ensures that all requirements are met. In addition, the constraint constructs the sort of the resulting lens snew. The formal definition for the constraints are shown in Figure 6.6.

As with the join lens, the Selectable constraint expands the lens sort and applies the components to the SelectableExp constraint, which performs the actual checks. The new lens sort is constructed by taking the components of the old lens, but extends the predicate with the user supplied predicate. The Simplify type family performs some simplifications for cosmetic purposes, such as short circuiting logical and expressions where one of the arguments is known.

The predicate should type check on the row type of the input lens. As the Haskell implementation makes use of hybrid predicates, it is sufficient to show that the static predicate $S$ satisfies the typing judgment $R \vdash S$ : bool in order to show that the underlying dynamic predicate $Q$ satisfies $R \vdash Q$ : bool. This can be checked with the TypesBool rt p constraint, where rt is the row type and p is the static predicate portion. In the constraint this check is performed by ensuring that Typ env phr ~ 'Just Bool is satisfied.

The next requirement is to check that the predicate does not refer to any columns that are in the outputs of the functional dependencies. The IgnoresOutputs pred fds constraint defined earlier performs this check.

The functional dependencies of the underlying lens must be in tree form, which is ensured by the constraint InTreeForm fds.

All required type class instances are summed up in the SelectImplConstraints constraint. As these are mainly required for technical reasons, the definition for this constraint is left out for brevity.

```
type SelectableExp p rt pred fds =
    (Typ env phr ~ 'Just Bool,
    IgnoresOutputs pred fds,
    InTreeForm fds,
    SelectImplConstraints rt p pred fds)
type Selectable p s snew =
    (snew ~ 'Sort (Ts s) (Rt s) (Simplify (p :& (P s))) (Fds s),
    SelectableExp p (Rt s) (P s) (Fds s))
```

Figure 6.6: The constraints required by the select lens.

Drop Lens The drop lens projects away a set of columns specified by the user. To construct a drop lens, the programmer provides a set of columns to drop with default values to use as well as a set of columns which determine the dropped columns. The dropped columns are specified using a type level row, where the key corresponds to the value, and the value is the default value.

The drop lens requires various checks on the underlying lens as well as the dropped columns and their default values. Recall the typing rule for the drop lens from Section 5 ,

$$
\begin{aligned}
& \text { T-Drop } \\
& F \equiv G \cup\{U \rightarrow \ell\} \quad \Gamma \vdash e: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P, F\right) \\
& U \subseteq \operatorname{dom}(R) \quad \cdot \vdash v: A \\
& \mathbf{L J D}[R,(\ell: A)](P) \quad \mathbf{D V}[R,(\ell: A)](P,(\ell=v)) \\
& \Gamma \vdash \operatorname{drop} \ell \text { determined by }(U, v) \text { from } e: \text { lens of }\left(\Sigma, R, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right) \\
& \text { dropl :: forall env (key :: [Symbol]) s snew. } \\
& \text { (Droppable env key s snew) => Lens s -> Lens snew }
\end{aligned}
$$

The side-conditions for the drop lens are checked using the Droppable env key snew constraint shown in Figure 6.7, where env is the set of dropped columns with their default values, key is the set of columns that must define the dropped column and the lens sort $s$. The components of the sort are projected out and checked using the DroppableExp constraint.

The first requirement is that the dropped columns should be in the domain of the input lens, and that the default values should also have the correct type. This is a

```
type DroppableExp env key rt pred fds rtnew =
    (EnvSubset (EvalRowType env) rt,
    Subset (Vars env) (TransClosure key fds),
    LJDI (Vars env) pred,
    DefVI env pred,
    DropImplConstraints env key rt pred fds rtnew)
type Droppable env key s snew =
    (snew ~ 'Sort (Ts s) (RemoveEnv (Vars env) (Rt s))
                            (Simplify (ReplacePredicate env (P s)))
            (DropColumn (Vars env) (Fds s)),
    DroppableExp env key (Rt s) (P s) (Fds s) (Rt snew))
```

Figure 6.7: The constraints required by the drop lens.
combination of the restrictions $U \subseteq \operatorname{dom}(R)$ and $\Gamma \vdash v: A$. The check is performed by calculating the row type of env using EvalRowType and then checking that it is a subset of $r t$.

The next check ensures that the functional dependencies of the underlying lens specify that the key columns define the dropped columns. This is checked by computing the transitive closure of key, and ensuring that the dropped columns are in the resulting set.

As covered in Section 4.2, the predicate on the underlying lens must pass both the lossless join and default value checks. These conditions are checked using the LJDI and DefVI type families defined in Section 6.2.3.

All further type class constraints required by the implementation are provided by the DropImplConstraints constraints. As these are mainly technical they have been left out for brevity.

The Droppable constraint also constructs the lens sort snew for the resulting lens. It removes all references to the old column in the new row type, the predicate of the lens as well as from the functional dependencies.

### 6.3.4 Lens Operations

Relational lenses form a bidirectional transformation on a relational database. Bidirectional transformations generally provide two operations. The forward transformation retrieving the view is called get, while the operation to submit changes is called put.

Get The get operation queries a database using the provided database connection for the view defined by the given lens. The view is a collection of records having the same record type as the lens. The get operation is executed in an IO monad, because the querying operation has side-effects. The LensGet constraint is a technical requirement providing the type class instance required by the implementation.

```
get :: forall t rt p fds. LensGet t rt p fds c =>
    c -> Lens t rt p fds -> IO (RecordsSet rt)
```

Put The programmer uses the put operation to alter the database. The programmer provides the operation with an updated view as a collection of records with the same record type that the lens has. The update operation is sideeffecting, so the put operation must be executed under an IO monad. The LensPut constraint also requires some type class constraints required by the implementation to be satisfied. The put function expects the set of records to satisfy the predicate and functional dependency constraints of the lens, but these are not statically checked. Instead the function returns a boolean value indicating if the operation was successful.

```
put :: forall c ts rt p fds. LensPut t rt p fds c =>
    c -> Lens t rt p fds -> RecordsSet rt -> IO Bool
```


### 6.4 Summary

For a programming language feature to become useful, it requires an implementation. Our first implementation of relational lenses in Links required feature specific extensions to the compiler. Such intrusive changes are reasonable to experiment with a language feature, but are not ideal in practice. The additional
rules increase the complexity of the compiler and are more likely to be incompatible with other language features.

Due to limitations in the type system, the Links implementation is also less flexible. The relational lens type checking rules require full knowledge of the lens type to be known. This prevents the programmer from using polymorphism on lens types, restricting their usage. The programmer is unable to define functions taking lenses with partially unknown sort types as arguments.

In this chapter we show how relational lenses can be implemented using the type-level programming features provided by Haskell. The relational lens library makes use of type classes, type families, data type promotion and qualified types. This implementation of relational lenses does not require any changes to the type system. The use of qualified types also allows the programmer to abstract over lenses, by allowing the type checking restrictions required to be required as a prerequisite to use the encompassing function.

The Haskell implementation also supports hybrid predicates. We show how the hybrid predicates introduced in Section 4.4.1 are implemented and provide an example of their usage. This allows the programmer to safely construct predicates at runtime.

Our work shows that qualified types and type level programming can be useful to ensure that any relational lens constructed is well-behaved. This is an improvement over the Links version, which requires trade-offs between static and dynamic predicates. The GHC extensions used by relational lenses are still not ideal however. Doing type-level programming feels mostly distinct from regular programming and we require many workarounds to achieve desired functionality such as missing let-in constructs. From this perspective a fully dependently typed language such as Idris 2 might be a better suited general purpose programming language for relational lenses.

Relational databases are ubiquitous in application development, yet working with them sometimes proves to be challenging. In the past, working with relational databases has justified the introduction of additional language features such as quasiquotation and anonymous functions in $\mathrm{C} \#$ [14]. We show that it is possible to implement relational lenses as a library that does not require changes to the compiler. In addition, this implementation makes relational lenses available in an
industrially-used programming language. We argue that our implementation is a good justification for introducing dependent types and type-level programming features to more programming languages.

## Chapter 7

## Related Work

### 7.1 Language-integrated query

Language-integrated query refers to programming language features that assist the programmer in expressing database queries from the host language. The idea is to let the programmer write queries in native language constructs, which can then be translated into the database query language, such as SQL, and efficiently executed on the database server [91, 82]. Language-integrated query expressions can be type-checked during compilation, ensuring that the application won't crash because of an ill-formed query when run. Language integrated-query supports more advanced queries than relational lenses, as there is no requirement for roundtripping guarantees. Unlike relational lenses, support for database updates with language integrated query can be considered an afterthought, and the user must manually translate changes made by the user into updates on the database.

The language integrated query can be preprocessed before being translated into the target query language. This allows the use of additional features that the query language does not support, as long as these can be desugared into the query language. An example is nested data structures, which can be desugared into multiple efficient queries [22]. More advanced data types such as tuples and records can also be used in predicates [25]. Predicates can also make use of functional abstractions, which can be removed using beta normalisation before being sent to the server. Our implementation of relational lenses does not support nested queries, but limited support may be feasible.

Most language-integrated query approaches allow the query to be expressed using list comprehensions [91. The C\# and F\# implementations also support constructing expressions using collection functions such as map and filter [82]. Relational lenses use lens composition instead, but it may be possible to extend the syntax to make use of list comprehensions.

From a language-integration perspective there are a different ways to support querying databases. The existing language-integrated query in Links generates the expression by translating the intermediate representation (IR) code used by the interpreter into an SQL expression directly [26]. The approach used by F\# and $\mathrm{C} \#$ is to translate the query expression into a query object constructing expression. The resulting query object which is created at runtime is then processed into an SQL query during execution 82. C\# uses meta-programming techniques to support the translation of functions into the underlying query language. Specifically, any lambda function expression can be cast to an Expression value representing that function. Meta-programming and reflection have the benefit of being useful beyond relational lenses. Our relational lenses in Links is implemented by extending the type-system, while the Haskell implementation uses data type promotion.

### 7.2 Updatable views and lenses

Updatable views have been studied extensively in database literature, and are supported (in very limited forms) in recent SQL standards and systems. We refer to Bohannon et al. [12] for discussion of earlier work on view updates and how relational lenses improve on it. Although updatable views (and their limitations) are well-understood, they are still finding applications in current research, for example for annotation propagation [17] or to "explain" missing answers, via updates to the source data that would cause a missing answer to be produced [47]. Date [28] discusses current practice and proposes pragmatic approaches to view update.

To the best of our knowledge, the work that comes closest to implementing relational lenses is Brul [93], which builds on top of BiGUL [58], a put-oriented language for programming bidirectional transformations. Brul includes the core relational lens primitives and these can be combined with other bidirectional
transformations written in BiGUL. However, Zan et al. 93 implement the statebased definitions of relational lenses over Haskell lists and do not evaluate their performance over large databases or consider efficient (incremental) techniques. They also do not consider functional dependencies or predicate constraints, so it is up to the programmer to ensure that these constraints are checked or maintained. Ko and Hu [57] recently proposed a Hoare-style logic for reasoning about BiGUL programs in Agda, which could perhaps be extended to reason about relational lenses.

Asano et al. [7] extend the putback-based approach for relational databases. They introduce a language for specifying put expressions, which are used to derive the corresponding get expression. A put expression can either verify its equality to a query of a database using a check expression, or use the data to update tables. Put expressions can be split horizontally or vertically to perform operations on multiple tables. The system does not perform static verification of the update operations, and does not verify functional dependencies or predicates. Put operations are total, and may fail if a check expression is not satisfied. Tran et al. [83] extend on this approach by showing how fine-grained put expressions can be defined in Datalog. These put-expressions can then be verified to ensure wellbehavedness. The system supports additional constraints, but there are some limitations and functional dependencies are not explicitly demonstrated. While the relational lens expressions are mostly derived from the forward semantics, adding additional lenses with different put semantics is possible. These putback approaches could be useful for deriving new lenses.

Object-relational mapping (ORM) is a popular technique for accessing and updating relational data from an object-oriented language. ORM can impose performance overhead but Bernstein et al. [9] show that incremental query compilation is effective in this setting. We would like to investigate whether incremental relational lenses could be composed with more conventional (edit) lenses to provide ORM-like capabilities for functional languages.

Bidirectional approaches to query languages for XML or graph data models have also been proposed [48, 65]. However, to the best of our knowledge these approaches are not incremental and have not been evaluated on large amounts of data. There is also work on translating updates to XML views over relational data, for example Fegaras [34]; however, this work does not allow joins in the
underlying relations.

### 7.3 Incremental computation

Incremental view maintenance is a well-studied topic in databases [44]. We employ standard incrementalisation translations for relations (sets of tuples) [73, 42]. More recently, Koch [59] developed an elegant framework for incremental query evaluation for bags (multisets of tuples), and Koch et al. [60] extended this approach to nested relational queries. We think it would be very interesting to investigate (incremental) lenses over nested collections or multisets.

Incremental recomputation also has a large amount of literature, including work on adaptive functional programming and self-adjusting computation [3, 46]. While closely related in spirit, this work focuses on a different class of problems, namely recomputing computationally expensive results when small changes are made to the inputs. In this setting, recording a large trace that caches intermediate results can yield significant savings if the small changes to the input only lead to small changes in the trace. It is unclear that such an approach would be effective in our setting. In any case, to the best of our knowledge, this approach has not been used for database queries or view updates.

Our approach to incrementalisation does draw inspiration from the incremental lambda calculus of Cai et al. [18]. They used Koch's incremental multiset operations in examples, but our set-valued relations and deltas also fit into their framework. Another relevant system, SQLCache [79], shows the value of language support for caching: in SQLCache, query results (and derived data) are cached and when the database is updated, dynamic checking is used to avoid recomputing results if the query did not depend on the changed data. Otherwise, SQLCache recomputes the results from scratch. Language support for incremental query evaluation could be used to improve performance in this case.

### 7.4 Incremental lenses

Wang et al. [89 considered incremental updates for efficient bidirectional programming over tree-shaped data structures, but not relations. There are several approaches to lenses that are based on translating changes. Edit lenses [50] are a
form of bidirectional transformations where, rather than translating directly between one data structure and another, the changes to a data structure are tracked and then translated into changes on the other data structure. Delta lenses [31] split the process of incremental view updates into an alignment and a propagation phase. The alignment computes the delta, which is not unique to the input and may be altered by the programmer. The delta is then translated into a delta on the opposite data structure using the propagation process. Diskin et al. [31] also consider undoability and invertibility for their lenses. They provide a general framework which is not based on differentiation and that does not rely on a specific data structure. Additional work also includes c-lenses 52 and update lenses [4].

Lenses translating changes between data structures are particularly useful in the case of symmetric lenses [32]. There, neither of the data structures contain all of the data, and thus none of the data structures can be considered the 'source' [49]. Changes could be described by insert, update and delete commands, and will usually result in similar insert, update or deletion commands for the other data structure.

None of these approaches has been applied to relational lenses as far as we know. Rather than utilize (and recapitulate) the needed technical background for these approaches, we have opted for a concrete approach based on incrementalisation by differentiation in the style of Cai et al. [18], but it would be interesting to understand the precise relationships among these various formalisms. The approach presented in this thesis preserves the exact semantics of the non-incremental version, and the delta of two tables is unique.

### 7.5 Row Type Inference

Records have long been a desirable feature to have in programming languages. They make it easier to work with data by allowing components in product types to be addressed by name rather than position. Records are of particular interest in the context of relational lenses. Not only do relational lenses require records to represent table records, but they also represent a challenging feature to integrate into programming language.

A straightforward method to implement records could be to define these as map-
pings from labels to values. The type of such a record would then be a mapping from labels to types. Such records are fairly straightforward, but also have their limitations. Each record that is constructed in such a way must be constructed with all the fields defined, and the corresponding type of the record must also reflect all fields. A function using such a record cannot accept a record which contains additional fields. This becomes troubling in combination with type inference. Consider a function $\lambda r$. r. $x$ which may receive the type $\Lambda \alpha .\{x: \alpha\} \rightarrow \alpha$ could not be used on a record $\{x$ : int, $y$ : int $\}$ because the two record types cannot be unified.

An approach to solving this problem is the use of row polymorphism [76, 54]. Row are constructed inductively, with the possibility of extending an existing row kind with an additional field. Each additional field is explicitly marked as either absent or present with a type $\tau$ describing the value in that field. Types are able to use polymorphism to abstract over a row kind. This allows various more interesting applications, such as requiring a record to contain some specific fields, along with arbitrary additional fields. A record field can also be explicitly removed by overriding it with an absent value. Additional work extends on row polymorphism to also support row concatenation [75, 88].

Such approaches to records also come with complexity. Records aren't just naively typed as mappings from fields to types, but instead require presence kinds. Type inference must be extended with additional unification rules to support row polymorphism. As more features such as row concatenation are added, the complexity is increased.

Another method of implementing records is by using constraints. A constraint on a record type can specify properties such as subset relations and disjointedness [70]. Such an approach can then support explicit row kinds that are closed mappings from labels to types, while still supporting more advanced operations. For example the type $\forall r, s, t . r \oplus s \sim t \Rightarrow \Pi r \rightarrow \Pi s \rightarrow \Pi t$ could be the type of a concatenation function, where $r \oplus s \sim t$ is a constraint specifying that the concatenation of row variables $r$ and $s$ must be equal to the row variable $t$. This approach can become cumbersome when a function requires many constraints on records, and in some cases an approach using both constraints and a specific row notation may be desirable.

There is no particular advantage to our implementation of records over other implementations such as HList [55] or CTRex. Our implementation is mainly included as an exercise to demonstrate how the type-level programming features in Haskell can be used to implement something like records.

### 7.6 Dependently Typed Programming

Haskell provides a limited form of dependent types [67, 43, 33]. Our relational lenses library makes extensive use of many related features such as generalized algebraic datatypes [61, 19, 85] type families [77, 78] and data type promotion [92]. These features are useful for performing additional type checking otherwise not possible, but cannot be used to prove the correctness of properties quantified over type variables.

In contrast, fully dependently typed programming languages, such as Agda [13], Coq [23] and Idris [15] can be used as general purpose theorem provers. These languages are based on the intuitionistic type theory of Martin-Löf and Sambin [66]. Based on this logic, the existence of an instance of a data type can be used as evidence that properties hold, and lemmas can be described by the type of a function. In the following snippet we define a data type $n<m$, which specifies that $n$ is less than $m$. The data type has two constructors, <base specifies that any number $n$ is less than its successor $n+1$. The other constructor <suc allows us to derive $n<m+1$ if we can provide evidence that $n<m$. Using this Lemma we can prove transitivity of the $<$ property, by defining a total function <trans that provides evidence that $n<o$ given evidence that $n<m$ and $m<o$. The function is implemented by pattern matching on the proof object $m<0$.

```
data _<_ : Nat -> Nat -> Set where
    <base : forall {n} -> n < S n
    <suc : forall {n m} -> n < m -> n < S m
<trans : forall {n m o} -> n < m -> m < o -> n < o
<trans pl <base = <suc pl
<trans p1 (<suc p2) = <suc (<trans p1 p2)
```

An advantage of fully dependently typed programming languages is that there is not as much of a distinction between types used to construct values and higher order kinds to be used within other types. There is also no distinction between functions and type families. As a result it is possible to use existing functions
and control structures such as $i f$-statements or let-binding for type-level computations. The type system also does not require constraints to add this type-level computation. The type system does not distinguish between string values and symbols, or between numbers and type level natural numbers. These changes make type-level programming feel more natural.

Scala 3 includes support for match types, offering a similar feature set to type families, but with additional support for sub-typing [11]. These features are used to ensure type-safety of tensor shapes in a TensorFlow programming interface. The type-level programming capabilities provided by Scala 3 would be helpful for implementing type-safe relational lenses, but it would be necessary to determine if the use of other features such as data type promotion and constraints can be translated into Scala.

The most commonly used programming languages today, with the exception of Haskell and Scala, do not support any form of dependent types. One could speculate that this may either be related to the complexity of dependent types or, because many of these languages are primarily designed as proof assistants. Rather than writing the whole application in such languages some, such as Coq, allow code to be extracted into other languages [62].

Idris 2 introduces quantitative type theory to a dependently typed programming language [16]. In quantitative type theory, variables can be given a multiplicity indicating how often they are used. A multiplicity of zero prevents variables from being used in any other runtime value. By giving proof objects a multiplicity of zero, it becomes safe to erase them during compilation, improving performance and bringing dependently typed programming closer to common applications.

## Chapter 8

## Future Work

### 8.1 Table Keys

Many relational database servers support auto-incrementing columns. Autoincrementing columns allow a record to be inserted into a table without specifying what the id value is. The database server then uses some mechanism to determine a value for the unspecified id column, so that it can be uniquely determined. In Postgresql for example, the programmer defines a column with the datatype SERIAL, which automatically creates a sequence. The sequence is incremented each time a new record is inserted into the table.

The Links implementation of relational lenses contains some support for autoincrementing columns. Links defines a variant datatype Serial, which can either be an existing key value $v$ in the database Key v , a new key that does not exist in the database yet and is considered unequal to any other key value NewKey, and a new key value NewKeyMapped i with an attached value $i$ which should be equal to any other new key value with the same $i$.

The updating process of a view $v$ then happens in a few stages. The first step is to replace any NewKey value with a value NewKeyMapped i such that the $i$ value is unique to the view. Relational lenses then compute the updated database table or change set to the database depending on which semantics are used. The resulting view or delta is used to update the database. While updating the database, the database returns the newly inserted values for each inserted record. This information is used to compute a mapping $m$ from NewKeyMapped $i$ values to
incremented values $j$.
Without further handling this approach violates the PutGet law, because the database has no notion of NewKeyMapped values. To restore well-behavedness, it is necessary to adjust the law. We assume that the put function returns a tuple of both the updated source table, and the mapping $m$ from NewKeyMapped values. We can then formulate PutGet as follows, where $m(Y)$ applies the mapping $m$ to each NewKeyMapped value in the view and the fix $(Y)$ function replaces all NewKey values with NewKeyMapped values.

$$
\begin{aligned}
& \text { let } Y^{\prime}=f i x(Y) \text { in } \\
& \text { let }\left(X^{\prime}, m\right)=\operatorname{put}\left(X, Y^{\prime}\right) \text { in } \\
& m\left(\operatorname{get}\left(X^{\prime}\right)\right)==Y^{\prime}
\end{aligned}
$$

We don't provide any further proofs that this approach works, and leave further investigation as future work.

### 8.2 Incremental Performance

The incremental semantics presented in Chapter 3 offer a significant improvement over the naive relational lens semantics. Unfortunately, these semantics are still not always ideal. All lenses currently require some intermediate queries to the database server, which slows down the update operation significantly.

Another issue is that join and select lenses can potentially result in very large intermediate queries that can seriously impact update performance. These happen in cases when views have functional dependencies $X \rightarrow Y$ such that there are many records with the same $X$ value. This would likely also result in very big deltas if the $Y$ value of the record is changed.

A potential remedy for the second issue may be to use an approach inspired by query shredding [22]. Rather than always querying the complete set of information required, the semantics may try to issue multiple queries for each functional dependency. The amount of data sent by the individual queries would then be smaller than the join of the queries containing redundant data.

Similarly, it could be helpful to investigate other change structure definitions. The current semantics define a delta as a tuple of the inserted records and deleted
records. This representation may include unnecessary information. When changing a field in a record for example, all the unchanged fields are also included. Using a different change structure may also reduce the amount of information that must be queried from the database server.

There are probably also many opportunities to improve the performance of the update semantics when certain criteria are met. For example, a select lens with only a single functional dependency $X \rightarrow Y$ that only filters records based on their $X$ value may not require additional queries for put operations. For deltas that only delete or update existing records (not inserting new ones), the numbers of queries that must be issued could potentially be reduced.

### 8.3 Concurrent Database Access

We have not yet investigated how well the current incremental semantics would handle multiple simultaneous updates to the same table at once. When working with databases, one typically uses transactions to detect when another process might interfere with the current process [45]. Transactions record which data is read from and written to during transaction processes. If there is some potentially harmful overlap between two transactions such as read-write or write-write conflicts, the transaction is rejected and all changes are rolled back. Transactions could possibly be used to improve concurrent access by relational lenses.

Another issue is the way delta views are calculated. If a client fetches a view, and the underlying view is then changed by another process while the client is altering their copy of the view, then when the client submits the changed view it will overwrite all of the changes by the other process. This could potentially be improved by ensuring that the delta is calculated by comparing it to a cached version of the old view. Another issue is that the granularity of deltas used by our incremental relational lenses is on a row-by-row basis. It may be helpful for the change structure to track exactly which columns or functional dependencies have been changed. These problems commonly affect lenses and distributed systems.

### 8.4 Additional Lenses

In this thesis we use the relational lens primitives introduced by Bohannon et al. [12]. These offer a good foundation of operations that the programmer might require, but other lenses may be of interest as well. The following lenses may also be interesting to investigate:

The join lens described here only supports the natural join, requiring a record in the output to be combined from records in both the left table and the right table. SQL also provides outer joins, where records are still included in the output if they are only contained in either the left or right table depending on the exact query. Such outer join semantics may have favourable performance to the inner join lenses. However, outer joins usually rely on null values, which are not supported in Links.

Another interesting application of lenses could be the integration of temporal databases [41]. Temporal databases can be used to keep track of when a record is valid, allowing the table to have multiple valid versions at different times. A relational lens could be defined to construct the table at a specific time.

Another helpful feature could be access control annotations on the source tables. The programmer could mark which fields may be written to and which fields should remain read-only. These permissions could be propagated to the views, preventing undesirable changes to the view. The read-only information might simplify put operations.

### 8.5 Well behaved Lenses

Relational lenses currently require well-behavedness. While this property is generally desirable, there are also some situations where this contradicts the requirements of the lenses. Consider the case where a select lens should be used to find records that contain some issue to be fixed. The select lens would use a predicate $P$ that describes the issue, returning a collection of records satisfying $P$. By fixing the records, it is assumed that they would not satisfy $P$ anymore, and it would not be possible to update the database with put, as this would violate PutGet. In addition to weakening well-behavedness it is also possible to strengthen it. Foster et al. [36] define a lens as very well behaved if it satisfies the PutPut law,
defined as:

$$
\operatorname{put}\left(X^{\prime}, \operatorname{put}(X, Y)\right)=\operatorname{put}\left(X^{\prime}, Y\right) .
$$

The PutPut rule specifies that the result of doing a put can be undone with another put, and is thus sometimes associated with undoability [80]. This property is often considered too restrictive [36, 80]. Relational lenses only satisfy a restricted version of this property, where PuTPut is only satisfied with monotonically decreasing views [12]. Diskin et al. [31] define undoability and invertability properties for delta-lenses, which are less restrictive. Investigating the applicability of such additional properties is left as future work.

## Chapter 9

## Conclusion

View update is a classical problem in databases. It has applications to database programming, security, and data synchronization. Updatable views seem particularly valuable in web programming settings, for bridging gaps between a normalized relational representation of application data and the representation the programmer actually wants to work with. Updatable views were an important source of inspiration for work on lenses in the functional programming languages community. There has been a great deal of research on lenses for functional programming since the influential work of Foster et al. [36], but relatively little of this work has found application to the classical view update problem. The main exception to this has been Bohannon et al. [12, who define well-behaved relational lenses based on a type system that tracks functional dependencies and predicate constraints in addition to the usual type constraints. Unlike updatable views in mainstream relational databases, relational lenses support complex view definitions (including joins) and offer strong guarantees of correct round-tripping behaviour. However, prior to the work on this thesis no public implementation of relational lenses was available. This thesis shows that relational lenses can be implemented in practice.

Turning relational lenses into a practical language feature poses some additional challenges. A naive implementation of the semantics as defined by Bohannon et al. [12] recomputes the entire table. This model may work on small databases, but typical web applications accumulate data over time. The growing database would slow down ever more as time progresses. This is particularly suboptimal
because web applications are designed to serve as many clients at the same time as possible.

This thesis introduces an equivalent definition for the semantics of relational lenses, based on incrementalization. Here, we draw on parallel developments in the database and functional programming communities: incremental view maintenance is a classical topic in databases, and there has been a great deal of work in the programming language community on adaptive or incremental functional programming. We show how to embed relational lenses (and their associated type and constraint system) into Links and prove the correctness of incremental versions of the select, drop, and join relational lenses and their compositions. We also present an implementation and evaluate its efficiency. In particular, we show that the naive approach of shipping the whole source database to a client program, evaluating the put operation in-memory, and replacing the old source tables with their new versions is realistic only for trivial data sizes. We demonstrate scalability to databases with hundreds of thousands of rows; for reasonable view and delta sizes, our implementation takes milliseconds whereas the naive approach takes seconds.

The other challenge to implementing language integrated relational lenses is the complexity of checking the typing rules required by relational lenses. For relational lenses to be well-behaved, it is necessary for the lenses to satisfy many side conditions. These side conditions are defined on the record type, functional dependency and predicate constraints applying to the input and output views for each lens primitive. The typing rules also ensure linear usage of the underlying tables.

As defined by Bohannon et al. [12], predicates are abstract sets and the checks on predicates are specified using relational algebra expressions. This thesis introduces a concrete predicate syntax amenable to implementation. We show how these predicates relate to abstract sets, and provide equivalent checks for our concrete predicate syntax. The work is supported by proofs verifying that these checks are sufficient. This thesis compares the advantages and disadvantages of performing the predicate checks during program execution or as a static check during compilation. The thesis then introduces hybrid predicates, which capture some benefits of both approaches. Such predicates allow portions of a static predicate to be erased in such a way that only the typing information retained when
this is sufficient to perform the predicate checks.
Bohannon et al. [12] define relational lenses as bidirectional transformations between database schemas. In the language integrated form of relational lenses, a more favourable representation would be lenses representing a handle to single view, such that the get and put functions are side-effecting operations on the lenses. This thesis presents the typing rules of such a full integration of relational lenses together with the work on predicates. The provided proofs show that lenses satisfying these typing rules produce an equivalent well-behaved database schema mapping relational lens.

The work presented in this thesis is used by two implementations of relational lenses. The first implementation extends the Links compiler, and has support for both static and dynamic predicates as well as incremental relational lens semantics. The other implementation of relational lenses is in the form of a Haskell library. This library can be implemented using various Haskell type-level programming techniques and does not require any changes to the compiler. This thesis documents how the relational lens typing rules are implemented in Haskell using type classes, type families, data type promotion and constraints.

Our work establishes for the first time the feasibility of relational lenses for solving classical view update problems in databases. We improve the performance of the relational lens semantics using incrementalization, provide a concrete predicate syntax and corresponding checks satisfying the specifications of abstract set predicates and provide a formulation of relational lenses that is more suitable for language integration. The resulting implementations support our hypothesis that relational lenses can be turned into a practical language feature.

## Bibliography

[1] Elm: A delightful language for reliable webapps. http://www.elm-lang.org, 2019.
[2] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. Addison Wesley, 1995.
[3] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. ACM Transactions on Programming Languages and Systems (TOPLAS), 28(6):990-1034, 2006.
[4] D. Ahman and T. Uustalu. Coalgebraic update lenses. Electronic Notes in Theoretical Computer Science, 308:25-48, 2014.
[5] T. Altenkirch and J. Chapman. Big-step normalisation. Journal of Functional Programming, 19(3-4):311-333, 2009.
[6] W. W. Armstrong. Dependency structures of data base relationships. In Proceedings of IFIP congress, volume 74, pages 580-583. Geneva, Switzerland, 1974.
[7] Y. Asano, S. Hidaka, Z. Hu, Y. Ishihara, H. Kato, H.-S. Ko, K. Nakano, M. Onizuka, Y. Sasaki, T. Shimizu, et al. A view-based programmable architecture for controlling and integrating decentralized data. arXiv preprint arXiv:1803.06674, 2018.
[8] F. Bancilhon and N. Spyratos. Update semantics of relational views. ACM Transactions on Database Systems (TODS), 6(4):557-575, 1981.
[9] P. A. Bernstein, M. Jacob, J. Pérez, G. Rull, and J. F. Terwilliger. Incremental mapping compilation in an object-to-relational mapping system. In Proceedings of the ACM SIGMOD International Conference on Manage-
ment of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, pages 1269-1280, 2013. doi: $10.1145 / 2463676.2465294$.
[10] G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C\#. In Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications, pages 479-498, 2007.
[11] O. E. P. Blanvillain. Abstractions for Type-Level Programming. PhD thesis, EPFL, 2022.
[12] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In Proceedings of the 25th ACM SIGMOD-SIGACTSIGART symposium on Principles of Database Systems, pages 338-347. ACM, 2006.
[13] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda - a functional language with dependent types. In International Conference on Theorem Proving in Higher Order Logics, pages 73-78. Springer, 2009.
[14] D. Box and A. Hejlsberg. LinQ: .NET language-integrated query. $M S D N$ Developer Centre, 89, 2007.
[15] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of functional programming, 23(5):552593, 2013.
[16] E. Brady. Idris 2: Quantitative type theory in practice. arXiv preprint arXiv:2104.00480, 2021.
[17] P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 150158, 2002.
[18] Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann. A theory of changes for higher-order languages: incrementalizing $\lambda$-calculi by static differentiation. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09-11, 2014, pages 145-155, 2014. doi: 10.1145/2594291.2594304.
[19] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.
[20] J. Cheney, S. Lindley, and P. Wadler. A practical theory of languageintegrated query. In Proceedings of the 18th ACM SIGPLAN international conference on Functional programming, ICFP '13, pages 403-416, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365. 2500586. URL http://doi.acm.org/10.1145/2500365.2500586.
[21] J. Cheney, S. Lindley, G. Radanne, and P. 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, 2014.
[22] J. Cheney, S. Lindley, and P. Wadler. Query shredding: efficient relational evaluation of queries over nested multisets. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pages 10271038, 2014.
[23] A. Chlipala. Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant. MIT Press, 2013.
[24] A. 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, Mumbai, India, January 15-17, 2015, pages 153-165, 2015. doi: 10.1145/2676726.2677004.
[25] E. Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In International Symposium on Database Programming Languages, pages 36-51. Springer, 2009.
[26] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures, pages 266-296, 2006. doi: 10.1007/ 978-3-540-74792-5_12. URL https://doi.org/10.1007/978-3-540-74792-5 12.
[27] G. Copeland and D. Maier. Making Smalltalk a database system. In Proceed-
ings of the 1984 ACM SIGMOD international conference on Management of data, pages 316-325, 1984.
[28] C. J. Date. View updating and relational theory. O'Reilly, 2012.
[29] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. ACM Transactions on Database Systems (TODS), 7(3): 381-416, 1982.
[30] C. Delobel and R. G. Casey. Decomposition of a data base and the theory of boolean switching functions. IBM Journal of Research and Development, 17(5):374-386, 1973.
[31] Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. Journal of Object Technology, 10:6: 1-25, 2011. doi: 10.5381/jot.2011.10.1.a6. URL https://doi.org/10.5381/jot.2011.10.1.a6
[32] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From state-to delta-based bidirectional model transformations: The symmetric case. In International Conference on Model Driven Engineering Languages and Systems, pages 304-318. Springer, 2011.
[33] R. A. Eisenberg. Dependent types in Haskell: Theory and practice. University of Pennsylvania, 2016.
[34] L. Fegaras. Propagating updates through XML views using lineage tracing. In Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA, pages 309320, 2010. doi: 10.1109/ICDE.2010.5447896. URL https://doi.org/10.1109/ ICDE. 2010.5447896.
[35] S. Fehrenbach and J. Cheney. Language-integrated provenance by trace analysis. In Proceedings of the 17 th ACM SIGPLAN International Symposium on Database Programming Languages, pages 74-84, 2019.
[36] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems (TOPLAS), 29(3):17, 2007.
[37] J. N. Foster, B. C. Pierce, and S. Zdancewic. Updatable security views. In Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE, pages 60-74. IEEE, 2009.
[38] N. Foster, K. Matsuda, and J. Voigtländer. Three complementary approaches to bidirectional programming. In Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures, pages 1-46, 2010. doi: 10.1007/978-3-642-32202-0_1. URL https://doi.org/10.1007/978-3-642-32202-0_1.
[39] S. Fowler, S. Harding, J. Sharman, and J. Cheney. Cross-tier web programming for curated databases: A case study. Under review., 2020. URL http://arxiv.org/abs/2003.03845.
[40] T. Freeman and F. Pfenning. Refinement types for ML. In Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, pages 268-277, 1991.
[41] V. Galpin and J. Cheney. Curating Covid-19 Data in Links. In Provenance and Annotation of Data and Processes, pages 237-243. Springer, 2020.
[42] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. IEEE Transactions on Knowledge and Data Engineering, 9(3):508-511, 1997.
[43] A. M. Gundry. Type inference, Haskell and dependent types. PhD thesis, University of Strathclyde, 2013.
[44] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. IEEE Data Engineering Bulletin, 18(2):3-18, 1995. URL http://sites.computer.org/debull/95JUN-CD.pdf.
[45] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. ACM computing surveys (CSUR), 15(4):287-317, 1983.
[46] M. A. Hammer, Y. P. Khoo, M. Hicks, and J. S. Foster. Adapton: composable, demand-driven incremental computation. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09-11, 2014, pages 156-166, 2014.
doi: 10.1145/2594291.2594324. URL http://doi.acm.org/10.1145/2594291. 2594324.
[47] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. A survey on provenance: What for? what form? what from? The International Journal on Very Large Data Bases, 26(6):881-906, 2017. doi: 10.1007/s00778-017-0486-1. URL https://doi.org/10.1007/s00778-017-0486-1.
[48] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010, pages 205-216, 2010. doi: 10.1145/ 1863543.1863573. URL http://doi.acm.org/10.1145/1863543.1863573.
[49] M. Hofmann, B. Pierce, and D. Wagner. Symmetric lenses. In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, pages 371-384, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926428. URL http://doi.acm.org/10.1145/1926385.1926428.
[50] M. Hofmann, B. Pierce, and D. Wagner. Edit lenses. In Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, pages 495-508, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103715. URL http://doi.acm.org/10.1145/2103656.2103715.
[51] R. Horn, S. Fowler, and J. Cheney. Language-integrated updatable views. In 31st Symposium on Implementation and Application of Functional Languages, 2020.
[52] M. Johnson and R. Rosebrugh. Delta lenses and opfibrations. In Proceedings of the Second International Workshop on Bidirectional Transformations: BX 2013, volume 57, pages 1-18. European Association of Software Science and Technology, 2013.
[53] M. P. Jones. A theory of qualified types. In European symposium on programming, pages 287-306. Springer, 1992.
[54] M. P. Jones and S. Peyton Jones. Lightweight extensible records for Haskell. In Haskell Workshop. Citeseer, 1999.
[55] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell, pages 96-107, 2004.
[56] C. Kiss, T. Field, S. Eisenbach, and S. Peyton Jones. Higher-order typelevel programming in Haskell. Proceedings of the ACM on Programming Languages, 3(ICFP):1-26, 2019.
[57] H. Ko and Z. Hu. An axiomatic basis for bidirectional programming. Proceedings of the ACM on Programming Languages, 2(POPL):41:1-41:29, 2018. doi: 10.1145/3158129. URL http://doi.acm.org/10.1145/3158129.
[58] H.-S. Ko, T. Zan, and Z. Hu. Bigul: A formally verified core language for putback-based bidirectional programming. In Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '16, pages 61-72, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4097-7. doi: 10.1145/2847538.2847544. URL http://doi.acm. org/10.1145/2847538.2847544.
[59] C. Koch. Incremental query evaluation in a ring of databases. In Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 87-98. ACM, 2010.
[60] C. Koch, D. Lupei, and V. Tannen. Incremental view maintenance for collection programming. In Proceedings of the 35th ACM SIGMOD-SIGACTSIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pages 75-90, 2016. doi: 10. 1145/2902251.2902286. URL http://doi.acm.org/10.1145/2902251.2902286.
[61] D. Leijen and E. Meijer. Domain specific embedded compilers. ACM Sigplan Notices, 35(1):109-122, 1999.
[62] P. Letouzey. Extraction in Coq: An overview. In Conference on Computability in Europe, pages 359-369. Springer, 2008.
[63] M. Ley. DBLP: Some lessons learned. Proceedings of the VLDB Endowment, 2(2):1493-1500, 2009.
[64] S. Lindley and J. Cheney. Row-based effect types for database integration.

In Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation, pages 91-102, 2012.
[65] D. Liu, Z. Hu, and M. Takeichi. Bidirectional interpretation of XQuery. In Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007, pages 21-30, 2007. doi: 10.1145/1244381.1244386. URL http: //doi.acm.org/10.1145/1244381.1244386.
[66] P. Martin-Löf and G. Sambin. Intuitionistic type theory, volume 9. Bibliopolis Naples, 1984.
[67] C. McBride. Faking it: simulating dependent types in Haskell. Journal of functional programming, 12(4-5):375-392, 2002.
[68] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In SIGMOD, 2006.
[69] R. Milner. A theory of type polymorphism in programming. Journal of computer and system sciences, 17(3):348-375, 1978.
[70] J. G. Morris and J. McKinna. Abstracting extensible data types: Or, rows by any other name. Proceedings of the ACM on Programming Languages, 3 (POPL):1-28, 2019.
[71] A. J. Pawson, J. L. Sharman, H. E. Benson, E. Faccenda, S. P. Alexander, O. P. Buneman, A. P. Davenport, J. C. McGrath, J. A. Peters, C. Southan, et al. The IUPHAR/BPS guide to PHARMACOLOGY: an expert-driven knowledgebase of drug targets and their ligands. Nucleic acids research, 42 (D1):D1098-D1106, 2014.
[72] T. Petricek. Data exploration through dot-driven development. In 31st European Conference on Object-Oriented Programming (ECOOP 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
[73] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. IEEE transactions on knowledge and data engineering, 3(3):337341, 1991. doi: 10.1109/69.91063. URL https://doi.org/10.1109/69.91063.
[74] R. Ramakrishnan and J. Gehrke. Database management systems (3. ed.). McGraw-Hill, 2003. ISBN 978-0-07-115110-8.
[75] D. Rémy. Typing record concatenation for free. In Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 166-176, 1992.
[76] D. Rémy. Type inference for records in a natural extension of ML. Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design, 1993.
[77] T. Schrijvers, M. Sulzmann, S. Peyton Jones, and M. Chakravarty. Towards open type functions for Haskell. Implementation and Application of Functional Languages, (12):233-251, 2007.
[78] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In Proceedings of the 13th ACM SIGPLAN international conference on Functional programming, pages 51-62, 2008.
[79] Z. Scully and A. Chlipala. A program optimization for automatic database result caching. In Proceedings of the 44 th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017, pages 271-284, 2017. URL http://dl.acm.org/citation.cfm?id= 3009891.
[80] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In International Conference on Model Driven Engineering Languages and Systems, pages 1-15. Springer, 2007.
[81] P. Stevens. A landscape of bidirectional model transformations. In International Summer School on Generative and Transformational Techniques in Software Engineering, pages 408-424. Springer, 2007.
[82] D. Syme. Leveraging .NET meta-programming components from F\#: integrated queries and interoperable heterogeneous execution. In $M L, 2006$.
[83] V.-D. Tran, H. Kato, and Z. Hu. Programmable view update strategies on relations. arXiv preprint arXiv:1911.05921, 2019.
[84] A. Ulrich and T. 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, Melbourne, Victoria, Aus-
tralia, May 31 - June 4, 2015, pages 1421-1426, 2015. doi: 10.1145/2723372. 2735359. URL http://doi.acm.org/10.1145/2723372.2735359.
[85] D. Vytiniotis, S. Weirich, and S. Peyton Jones. Simple unification-based type inference for GADTs. In International Conference on Functional Programming (ICFP'06), 2006.
[86] P. Wadler. Comprehending monads. In Proceedings of the 1990 ACM Conference on LISP and Functional Programming, pages 61-78, 1990.
[87] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 60-76, 1989.
[88] M. Wand. Type inference for record concatenation and multiple inheritance. Information and Computation, 93(1):1-15, 1991.
[89] M. Wang, J. Gibbons, and N. Wu. Incremental updates for efficient bidirectional transformations. In Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011, pages 392-403, 2011. doi: 10.1145/2034773.2034825. URL http://doi.acm.org/10.1145/2034773.2034825.
[90] E. Wcisło, P. Habela, and K. Subieta. A Java-integrated Object Oriented Query Language. In International Conference on Informatics Engineering and Information Science, pages 589-603. Springer, 2011.
[91] L. Wong. Kleisli, a functional query system. Journal of Functional Programming, 10(1):19-56, 2000.
[92] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation, pages 53-66, 2012.
[93] T. Zan, L. Liu, H. Ko, and Z. Hu. Brul: A putback-based bidirectional transformation library for updatable views. In Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software,

ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016., pages 77-89, 2016. URL http://ceur-ws.org/Vol-1571/paper_3.pdf.

## Appendix A

## Proofs for Chapter 2

## A. 1 Comparison to original Relational Lenses

In this section we compare the definition of the select lens as defined by Bohannon et al. [12] with our definition in Chapter 2.

## Original Formulation of Select Lens

In the original formulation by Bohannon et al. [12], a database $\varphi, \gamma$ is a mapping from relation names $S, T$ to relation names $M, N$. The sort of a relation name $S$, written $\operatorname{sort}(S)$, is a tuple $(R, P, F)$. For any mapping $\varphi(S)=M$ such that sort $(S)=(R, P, F)$, it is required that $M: \operatorname{Rel}(R, P, F)$. The schema of a database $\Delta, \Sigma$ is a set of relation names. A database $\varphi$ is an instance of a schema $\Delta$, written $\varphi: \Delta$ if it contains a mapping for each relation name in $\Delta$.

A relational lens consumes a relation in the left database, replacing it with a new relation in the right database. The following lens removes the relation $S$ and replaces it with a relation $T$, where $T$ only contains the entries from $S$ that satisfy the predicate $P$ :

```
select from S where P as T
```

We define $I$ as the above lens. The lens has the type $\{S\} \uplus \Sigma \Leftrightarrow\{T\} \uplus \Sigma$ where sort $(S)=(R, Q, F)$ and $\operatorname{sort}(T)=(R, P \wedge Q, F)$. The function $I \nearrow(\varphi)$ computes the get direction given a database $\varphi$ and $I \searrow(\gamma, \varphi)$ computes the put direction given the updated view database $\gamma$ and the original database $\varphi$.

$$
\begin{aligned}
I \nearrow(\varphi) & =\varphi \backslash_{S}[T \mapsto P \cap \varphi(S)] \\
I \searrow(\gamma, \varphi) & =\gamma \backslash_{T}\left[S \mapsto M_{1} \backslash N_{\#}\right]
\end{aligned}
$$

where

$$
\begin{aligned}
M_{1} & =(\neg P \cap \varphi(S)) \leftarrow_{F} \gamma(T) \\
N_{\#} & =\left(P \cap M_{1}\right) \backslash \gamma(T) \\
F & =f d(S)
\end{aligned}
$$

Bohannon et al. [12] show that $I \nearrow(\varphi)$ is a total function of type $(\{S\} \uplus \Sigma) \rightarrow$ $(\{T\} \uplus \Sigma)$, where sort $(S)=(R, Q, F)$ and $\operatorname{sort}(T)=(R, P \wedge Q, F)$. They also show that $I \searrow(\gamma, \varphi)$ is a total function of type $(\{T\} \uplus \Sigma) \times(\{S\} \uplus \Sigma) \rightarrow(\{S\} \uplus \Sigma)$.

## Sequential Select Lens

The select lens introduced in Chapter 2 is a bidirectional mapping between relation types $\operatorname{Rel}(R, Q, F) \Leftrightarrow \operatorname{Rel}(R, P \wedge Q, F)$. We derive the semantics of a select lens $v=\operatorname{select}_{P}: \operatorname{Rel}(R, Q, F) \Leftrightarrow \operatorname{Rel}(R, P \wedge Q, F)$ by constructing a relational lens $I=\operatorname{select}_{S} P$ as $T$ assuming that there are no other tables in the schema $(\Sigma=\varnothing)$. We then define the get and put functions by first constructing databases using the provided relations, then calling the underlying get or put function on $I$ and finally projecting the result onto the resulting table name. We swap the order of the put function arguments.

$$
\begin{aligned}
\operatorname{get}_{v}(M) & =I \nearrow(\{S \mapsto M\})(T) \\
\operatorname{put}_{v}(M, N) & =I \searrow(\{T \mapsto N\},\{S \mapsto M\})(S)
\end{aligned}
$$

The above expressions for get and put are simplified by substituting the arguments into the definitions of $I \nearrow$ and $I \searrow$. In the get function $\varphi$ is substituted with $\{S \mapsto M\}$. When expanded, the expression $\varphi(S)$ is equal to $M$. The resulting expression of the get function takes the form $\varphi_{\backslash S}[S \mapsto O](S)$, which is simplified to the expression $O$. The function body of $\operatorname{put}(M, N)$ similarly replaces each instance of $\varphi(S)$ with $M$ and $J(S)$ with $N$. The resulting expression of the form $\chi_{T}[S \mapsto O](S)$ can be simplified to $O$. The resulting $g e t_{v}$ and $p u t_{v}$ functions are:

$$
\begin{aligned}
\operatorname{get}_{v}(M)= & \sigma_{P}(M) \\
\operatorname{put}_{v}(M, N)= & \operatorname{let} M_{0}=\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), N\right) \text { in } \\
& \text { let } N_{\#}=\sigma_{P}\left(M_{0}\right)-N \text { in } \\
& M_{0}-N_{\#}
\end{aligned}
$$

The formulations is equivalent with the following notable differences:

- The use of let-bindings rather than where-bindings.
- The relational select operation $\left(\sigma_{P}(M)\right)$ is used instead of set intersection $(P \cap M)$. Semantically these are equivalent, but the relational select operator makes it more explicit that $P$ is fixed. This makes incrementalization easier, because the incremental version of the select operator is simpler than the incremental intersection operator.
- Set difference is denoted using - rather than $\backslash$, but rely on the same semantics of these operations.
- Functional notation for operators such as relational merge are written $M \leftarrow_{F}$ $N$ instead of merge $_{F}(M, N)$.
- Only the relevant tables are computed instead of the full database.

We know that $I \nearrow(\varphi)$ is a total function of the type $\{S\} \rightarrow\{T\}$, where sort $(S)=$ $(R, Q, F)$ and sort $(T)=(R, P \wedge Q, F)$. If $M$ is of type $\operatorname{Rel}(R, Q, F)$, then $\{S \mapsto M\}$ is a database of schema $\{S\}$ and $I \nearrow(\{S \mapsto M\})$ is a database instance of the schema $\{T\}$. The projection of this expression is a relation of type $\operatorname{Rel}(R, P \wedge$ $Q, F)$, making $g e t_{v}$ a total function from type $\operatorname{Rel}(R, Q, F) \rightarrow \operatorname{Rel}(R, P \wedge Q, F)$. $I \searrow(\varphi)$ is a total function from $\{T\} \times\{S\} \rightarrow\{S\}$, where sort $(S)=(R, Q, F)$ and sort $(T)=(R, P \wedge Q, F) .\{S \mapsto M\}$ and $\{T \mapsto N\}$ are instances of $\{S\}$ and $\{T\}$. It follows that $p u t_{v}$ is a total function of type $\operatorname{Rel}(R, Q, F) \times \operatorname{Rel}(R, P \wedge Q, F) \rightarrow$ $\operatorname{Rel}(R, Q, F)$.

We now show that the round-tripping rules hold, starting with GetPut:

$$
\begin{array}{rlrl}
\forall \varphi & :\{S\} .\{S \mapsto \varphi(S)\}=\varphi & \\
\text { get }_{v}(\operatorname{put}(M, N)) & \\
& =I \nearrow(\{S \mapsto I \searrow(\{T \mapsto N\},\{S \mapsto M\})(S)\})(T) & \text { def. get } t_{v} / \text { put }_{v} \\
& =I \nearrow(I \searrow(\{T \mapsto N\},\{S \mapsto M\}))(T) & I \text { well behaved }  \tag{1}\\
& =\{T \mapsto N\}(T) & \\
& =N & 1 \text { (1) }
\end{array}
$$

Next we show that PutGet holds:

$$
\operatorname{put}_{v}(M, \operatorname{get}(M))
$$

$$
\begin{array}{lr}
=v \searrow(\{T \mapsto v \nearrow(\{S \mapsto M\})(T)\},\{S \mapsto M\})(S) & \text { def. get } v_{v} / \text { put }_{v} \\
=v \searrow(v \nearrow(\{S \mapsto M\}),\{S \mapsto M\})(S) & \\
=\{S \mapsto M\}(S) & I \text { well behaved } \\
=M &
\end{array}
$$

The remaining lenses (with the exception of the change to the project lens shown in the following section) can be defined in a similar manner.

## A. 2 BPV to Structural Sequential Lenses

The representation of relational lenses presented in Section 2.2 differs slightly from the representation from Bohannon et al. [12], which we refer to as $B P V$ relational lenses. BPV lenses construct a bidirectional transformation between two database schemas, where each schema is a mapping from schema names to table types. Our representation is a mapping between structural schemas, which are product types of individual table types that aren't referred to by name. Consider the difference between the lens select $P_{P} S$ as $T$ which maps between the schemas $\{S\}$ and $\{T\}$ and the lens select ${ }_{P}$ which maps between the relations $\operatorname{Rel}(R, Q, F)$ and $\operatorname{Rel}(R, P \wedge Q, F)$. Our formulation does not refer to tables by relation names. BPV style lenses are used for Chapter 5 which also contains the corresponding typing rules.

BPV relational lenses must be careful to ensure linearity of relation names. This ensures that no lens consumes a lens twice, as the put operation would otherwise produce two variants of these lenses which may differ (e.g. if a table is joined with itself, and a record is then only deleted from the left lens). The structural sequential lenses avoid this issue, as there is no notion of reusing a table. Instead each of the tables in the product type are considered unique.

For any lens $I: \Sigma \Leftrightarrow \Delta$ we can compute the mapping $m$ from any $S \in \Delta$ to the corresponding sequential lenses using the seq function. We use the notation $m_{\backslash \vec{S}}$ to mean the mapping $m$ with all entries $\vec{S}$ removed, $m[S \mapsto I]$ to mean the mapping $m$ with $S$ set to $I$ and $m(S)$ a lookup of the value $S$ in $m$.

$$
\operatorname{seq}(I)=\operatorname{seq}^{\prime}\left(\left\{S \mapsto \mathrm{id}_{S} \mid S \in \Sigma\right\}, I\right)
$$

where seq' is defined as

$$
\operatorname{seq}^{\prime}\left(m, \operatorname{select}_{P} S \text { as } T\right)=m\left[T \mapsto m(S) ; \text { select }_{P}\right]
$$

$\operatorname{seq}^{\prime}(m$, drop $\ell$ determined by $(U, v)$ from $S$ as $T)=m[T \mapsto m(S)$;drop $\ell$ determined by $(U, v)]$

$$
\begin{aligned}
\operatorname{seq}^{\prime}\left(m, \operatorname{join}_{P_{d}, Q_{d}} S_{1} S_{2} \text { as } T\right) & =m\left[T \mapsto m\left(S_{1}\right) \otimes m\left(S_{2}\right) ; \operatorname{join}_{P_{d}, Q_{d}}\right] \\
\operatorname{seq}^{\prime}(m, I ; J) & =\operatorname{seq}^{\prime}\left(\operatorname{seq}^{\prime}(m, I), J\right)
\end{aligned}
$$

## A. 3 Equivalence of project lens definitions

Theorem 15. The definition of the project lens given in Section 2.3.3:

$$
\begin{aligned}
\operatorname{get}(M)= & \pi_{U-A}(M) \\
\operatorname{put}(M, N)= & \left.\boldsymbol{\operatorname { l e t }} M_{1}=N \bowtie\{\{A=a\}\}\right) \text { in } \\
& \text { revise }_{X \rightarrow A}\left(M_{1}, M\right)
\end{aligned}
$$

is equivalent to the definition given in Bohannon et al. [12]:

$$
\begin{aligned}
\operatorname{get}(M)= & \pi_{U-A}(M) \\
\operatorname{put}(M, N)= & \boldsymbol{\operatorname { l e t }} N_{\text {new }}=N-\pi_{U-A}(M) \text { in } \\
& \operatorname{let}_{M_{0}}=(M \bowtie N) \cup\left(N_{\text {new }} \bowtie\{\{A=a\}\}\right) \text { in } \\
& \text { revise }_{X \rightarrow A}\left(M_{0}, M\right)
\end{aligned}
$$

Proof.
$M: U$ and $N: U-A$ with $A \in U$ and $X \subseteq U-A$
$M_{0}=(M \bowtie N) \cup\left(\left(N-\pi_{U-A}(M)\right) \bowtie\{(A=a)\}\right)$
suppose ( $M, U, A, X$ )
$M_{1}=N \bowtie\{(A=a)\}$
Now show revise $X_{X \rightarrow A}\left(M_{0}, M\right)=$ revise $_{X \rightarrow A}\left(M_{1}, M\right)$.
$\Longrightarrow$ direction :
$m \in M_{0}$
suppose ( $m$ )
if $m \in M \bowtie N:$
$m \in M$
def. $\bowtie$
$m[U-A] \in N$
$m^{\prime}=m[U-A] \leftarrow(A=a)$
define $\left(m^{\prime}\right)$
$m^{\prime}[U-A]=m[U-A]$
$\hookrightarrow m^{\prime} \in M_{1}$
(1); def. $\ltimes$
$\hookrightarrow$ recrevise $_{X \rightarrow A}\left(m^{\prime}, M\right)=m^{\prime} \leftarrow m[A]$

$$
\begin{align*}
&=m \leftarrow m[A]  \tag{3}\\
&=\text { recrevise }_{X \rightarrow A}(m, M) \quad m \in M \text {; def. recrevise } \\
& X \rightarrow A
\end{align*}(\cdot, \cdot)
$$

$$
\hookrightarrow m \in M_{1} \quad \bowtie \text { monotone; (1) }
$$

$\Longleftarrow$ direction:

$$
\begin{align*}
& m \in M_{1} \\
& m[U-A] \in N
\end{align*} \quad \text { suppose ( } m \text { ) }
$$

## Appendix B

## Proofs for Chapter 3

## B. 1 Proofs for Section 3.1

## Proof of Theorem 2

Theorem 2. If $q: \operatorname{Rel}\left(U_{1}\right) \times \cdots \times \operatorname{Rel}\left(U_{n}\right) \rightarrow \operatorname{Rel}(U)$ then $\delta(q)$ and $(q)^{\dagger}$ are delta-correct with respect to $q$.

Proof. By induction on the structure of $q$. First observe that in any case $(q)^{\dagger}$ is deltacorrect with respect to $q$ if and only if $\delta(q)$ is. Thus, we show that $\delta(q)$ is delta-correct by induction, and the reasoning for $(q)^{\dagger}$ is similar.

- If $q=M$, a constant relation, then $\delta(q)=\varnothing$ which is delta-correct with respect to $q$ since $q\left(R_{1} \oplus \Delta R_{1}, \ldots, R_{n} \oplus \Delta R_{n}\right)=M=q\left(R_{1}, \ldots, R_{n}\right) \oplus \varnothing$. Minimality is obviously preserved.
- If $q=R_{i}$, a relation reference, then $\delta\left(R_{i}\right)=\Delta R_{i}$ is delta-correct with respect to $q$ since $q\left(R_{1} \oplus \Delta R_{1}, \ldots, R_{n} \oplus \Delta R_{n}\right)=R_{i} \oplus \Delta R_{i}=q\left(R_{1}, \ldots, R_{n}\right) \oplus \Delta R_{i}$. Minimality is obviously preserved.
- If $q=o p\left(q_{1}, \ldots, q_{n}\right)$ then the desired result follows from the definition of $\delta o p$ (which are delta-correct by construction), the induction hypothesis applied to the subexpressions $q_{i}$, and finally Lemma 25 .
- If $q=$ let $R=q_{1}$ in $q_{2}$, then the translation is

$$
\text { let }(R, \Delta R)=\left(q_{1}\right)^{\dagger} \text { in } \delta\left(q_{2}(R)\right)
$$

where $q_{2}$ has an additional parameter $R$, so $\left(q_{2}\right)^{\dagger}$ will have an additional pair of parameters $(R, \Delta R)$. By induction both $\left(q_{1}\right)^{\dagger}$ and $\left(q_{2}\right)^{\dagger}(R, \Delta R)$ are delta-correct with respect to $q_{1}$ and $q_{2}$ respectively. We reason as follows:

$$
\begin{align*}
& \operatorname{let} R^{\prime}=q_{1}\left(R_{1} \oplus \Delta R_{1}, \ldots, R_{n} \oplus \Delta R_{n}\right) \text { in } q_{2}\left(R^{\prime}, R_{1} \oplus \Delta R_{1}, \ldots, R_{n} \oplus \Delta R_{n}\right)  \tag{1}\\
= & \operatorname{let} R^{\prime}=q_{1}\left(R_{1}, \ldots, R_{n}\right) \oplus \delta(q)\left(\left(R_{1}, \Delta R_{1}\right), \ldots,\left(R_{n}, \Delta R_{n}\right)\right) \text { in } q_{2}\left(R^{\prime}, R_{1} \oplus \Delta R_{1}, \ldots, R_{n} \oplus \Delta R_{n}\right)  \tag{2}\\
= & \operatorname{let}\left(R^{\prime}, \Delta R^{\prime}\right)=\left(q_{1}\right)^{\dagger}\left(\left(R_{1}, \Delta R_{1}\right), \ldots,\left(R_{n}, \Delta R_{n}\right)\right) \text { in } q_{2}\left(R^{\prime} \oplus \Delta R^{\prime}, R_{1} \oplus \Delta R_{1}, \ldots, R_{n} \oplus \Delta R_{n}\right)  \tag{3}\\
= & \operatorname{let}\left(R^{\prime}, \Delta R^{\prime}\right)=(q)  \tag{4}\\
& q_{2}\left(\left(R^{\prime}, R_{1}, \ldots, R_{n}\right) \oplus \delta\left(R_{1}, \Delta R_{1}\right), \ldots,\left(\left(R_{n}, \Delta R_{n}\right)\right)\right. \text { in }  \tag{5}\\
= & \left(\operatorname{let}\left(R^{\prime}, \Delta R^{\prime}\right),\left(R_{1}, \Delta R_{1}\right), \ldots,\left(R_{n}, \Delta R_{n}\right)\right)  \tag{6}\\
& \oplus\left(q_{1}\right)^{\dagger}\left(\left(R_{1}, \Delta R_{1}\right), \ldots,\left(R_{n}, \Delta R_{n}\right)\right) \text { in } q_{2}\left(R^{\prime}, R_{1}, \ldots, \Delta R_{n}^{\prime}\right)=\left(q_{1}\right)^{\dagger}\left(\left(R_{1}, \Delta R_{1}\right), \ldots,\left(R_{n}, \Delta R_{n}\right)\right) \text { in } \delta\left(q_{2}\right)\left(\left(R^{\prime}, \Delta R^{\prime}\right),\left(R_{1}, \Delta R_{1}\right), \ldots,\left(R_{n}, \Delta R_{n} \backslash\right\rceil\right) \\
= & \left(\operatorname{let} R^{\prime}=q_{1}\left(R_{1}, \ldots, R_{n}\right) \text { in } q_{2}\left(R^{\prime}, R_{1}, \ldots, R_{n}\right)\right) \oplus \delta\left(\operatorname{let} R^{\prime}=q_{1}\left(R_{1}, \ldots, R_{n}\right) \text { in } q_{2}\left(R^{\prime}, R_{1}, \ldots, R_{n}\right)\right)
\end{align*}
$$

Moreover, minimality is preserved by $\delta\left(q_{1}\right)$, so $\Delta R^{\prime}$ is minimal, which together with the minimality of other deltas implies that the delta computed by $\delta\left(\right.$ let $R^{\prime}=$ $q_{1}$ in $\left.q_{2}\left(R^{\prime}\right)\right)$ is also minimal. This shows that $\delta\left(\operatorname{let} R^{\prime}=q_{1}\right.$ in $\left.q_{2}\left(R^{\prime}\right)\right)$ is deltacorrect.

## Proof of Lemma 26

Lemma 26. [Valid optimisations] Assume $\Delta M, \Delta N$ are minimal for $M, N$ respectively. Then:

1. $\dot{\sigma}_{P}(M, \Delta M)=\left(\sigma_{P}\left(\Delta M^{+}\right), \sigma_{P}\left(\Delta M^{-}\right)\right)$
2. $\dot{\pi}_{U}(M, \Delta M)=\left(\pi_{U}\left(\Delta M^{+}\right)-\pi_{U}(M), \pi_{U}\left(\Delta M^{-}\right)-\pi_{U}(M \oplus \Delta M)\right)$
3. $(M, \Delta M) \dot{\bowtie}(N, \Delta N)=\left(\left((M \oplus \Delta M) \bowtie \Delta N^{+}\right) \cup\left(\Delta M^{+} \bowtie(N \oplus \Delta N)\right),\left(\Delta M^{-} \bowtie\right.\right.$ $\left.N) \cup\left(M \bowtie \Delta N^{-}\right)\right)$
4. $\dot{\rho}_{A / B}(M, \Delta M)=\left(\rho_{A / B}\left(\Delta M^{+}\right), \rho_{A / B}\left(\Delta M^{-}\right)\right)$
5. If $N \subseteq M$ and $N \oplus \Delta N \subseteq M \oplus \Delta M$ then $(M, \Delta M) \dot{-}(N, \Delta N)=\Delta M \ominus \Delta N$

To simplify notation, we abbreviate $M^{\prime}=M \oplus \Delta M$ and $N^{\prime}=N \oplus \Delta N$. Recall that the positive and negative parts of a delta are always disjoint; we freely use the fact that $(X \cup Y)-Z=(X-Z) \cup Y$ when $Y$ and $Z$ are disjoint.

The proof of parts (1)-(4) all follow a similar pattern: we first find expressions $\Delta N^{+}, \Delta N^{-}$ for the components of a minimal relational delta such that $o p(M \oplus \Delta M)=o p(M) \oplus \Delta N$.

By Lemma 24 it then follows that $\delta o p(M, \Delta M)=o p(M \oplus \Delta M) \ominus o p(M)=(o p(M) \oplus$ $\Delta N) \ominus o p(M)=\Delta N$.

## Proof of part (1)

$$
\begin{aligned}
\sigma_{P}\left(M^{\prime}\right) & =\sigma_{P}\left(\left(M \cup \Delta M^{+}\right)-\Delta M^{-}\right) \\
& =\sigma_{P}\left(M \cup \Delta M^{+}\right)-\sigma_{P}\left(\Delta M^{-}\right) \\
& =\left(\sigma_{P}(M) \cup \sigma_{P}\left(\Delta M^{+}\right)\right)-\sigma_{P}\left(\Delta M^{-}\right) \\
& =\sigma_{P}(M) \oplus\left(\sigma_{P}\left(\Delta M^{+}\right), \sigma_{P}\left(\Delta M^{-}\right)\right)
\end{aligned}
$$

where clearly $\sigma_{P}\left(\Delta M^{+}\right) \cap \sigma_{P}\left(\Delta M^{-}\right)=\varnothing$. Moreover, $\sigma_{P}\left(\Delta M^{+}\right) \cap \sigma_{P}(M)=\varnothing$ by minimality of $\Delta M$, and likewise $\sigma_{P}\left(\Delta M^{-}\right) \subseteq \sigma_{P}(M)$ by minimality and monotonicity of selection.

## Proof of part (2)

First we observe that

$$
\begin{aligned}
\pi_{U}(M \cup N) & =\pi_{U}(M) \cup \pi_{U}(N) \\
\pi_{U}(M-N) & =\pi_{U}(M)-\left(\pi_{U}(N)-\pi_{U}(M-N)\right)
\end{aligned}
$$

Now we proceed as follows:

$$
\begin{aligned}
\pi_{U}\left(M^{\prime}\right) & =\pi_{U}\left(\left(M \cup \Delta M^{+}\right)-\Delta M^{-}\right) \\
& =\left(\pi_{U}\left(M \cup \Delta M^{+}\right)\right)-\left(\pi_{U}\left(\Delta M^{-}\right)-\pi_{U}\left(M^{\prime}\right)\right) \\
& =\left(\pi_{U}\left(\Delta M^{+}\right) \cup \pi_{U}(M)\right)-\left(\pi_{U}\left(\Delta M^{-}\right)-\pi_{U}\left(M^{\prime}\right)\right) \\
& =\left(\left(\pi_{U}\left(\Delta M^{+}\right) \cup \pi_{U}(M)\right)-\left(\pi_{U}(M)-\pi_{U}(M)\right)\right)-\left(\pi_{U}\left(\Delta M^{-}\right)-\pi_{U}\left(M^{\prime}\right)\right) \\
& =\left(\pi_{U}(M) \cup\left(\pi_{U}\left(\Delta M^{+}\right)-\pi_{U}(M)\right)\right)-\left(\pi_{U}\left(\Delta M^{-}\right)-\pi_{U}\left(M^{\prime}\right)\right) \\
& =\pi_{U}(M) \oplus\left(\pi_{U}\left(\Delta M^{+}\right)-\pi_{U}(M), \pi_{U}\left(\Delta M^{-}\right)-\pi_{U}\left(M^{\prime}\right)\right)
\end{aligned}
$$

where line 4 follows from the identity $X \cup(Y-Z)=(X \cup Y)-(Z-X)$, line 2 from the first observation above and line 1 by the second observation.

To establish minimality, clearly $\left(\pi_{U}\left(\Delta M^{+}\right)-\pi_{U}(M)\right) \cap \pi_{U}(M)=\varnothing$, while $\pi_{U}\left(\Delta M^{-}\right)-$ $\pi_{U}\left(M^{\prime}\right) \subseteq \pi_{U}(M)$ by monotonicity of projection since $\Delta M^{-} \subseteq M$.

## Proof of part (3)

We need to prove:

$$
M^{\prime} \bowtie N^{\prime}=\left(M \bowtie N \cup\left(M^{\prime} \bowtie \Delta N^{+} \cup \Delta M^{+} \bowtie N^{\prime}\right)\right)-\left(M \bowtie \Delta N^{-} \cup \Delta M^{-} \bowtie N\right)
$$

We first consider the special cases where $\Delta M^{-}=\Delta N^{-}=\varnothing$ and $\Delta M^{+}=\Delta N^{+}=\varnothing$. In the first case we have:

$$
\begin{aligned}
\left(M \cup \Delta M^{+}\right) \bowtie\left(N \cup \Delta N^{+}\right) & =M \bowtie N \cup\left(\Delta M^{+} \bowtie N \cup M \bowtie \Delta N^{+} \cup \Delta M^{+} \bowtie \Delta N^{+}\right) \\
& =M \bowtie N \cup\left(\Delta M^{+} \bowtie N \cup \Delta M^{+} \bowtie \Delta N^{+} \cup M \bowtie \Delta N^{+} \cup \Delta M^{+} \bowtie \Delta N^{+}\right) \\
& =M \bowtie N \cup\left(\left(M \cup \Delta M^{+}\right) \bowtie \Delta N^{+} \cup \Delta M^{+} \bowtie\left(N \cup \Delta N^{+}\right)\right)
\end{aligned}
$$

In the second case we have:

$$
\begin{aligned}
\left(M-\Delta M^{-}\right) \bowtie\left(N-\Delta N^{-}\right) & =M \bowtie\left(N-\Delta N^{-}\right)-\Delta M^{-} \bowtie\left(N-\Delta N^{-}\right) \\
& =M \bowtie N-M \bowtie \Delta N^{-}-\Delta M^{-} \bowtie\left(N-\Delta N^{-}\right) \\
& =M \bowtie N-\left(M \bowtie \Delta N^{-} \cup \Delta M^{-} \bowtie\left(N-\Delta N^{-}\right)\right) \\
& =M \bowtie N-\left(M \bowtie \Delta N^{-} \cup\left(\Delta M^{-} \bowtie N-\Delta M^{-} \bowtie \Delta N^{-}\right)\right) \\
& =M \bowtie N-\left(\left(M \bowtie \Delta N^{-} \cup \Delta M^{-} \bowtie N\right)-\left(\Delta M^{-} \bowtie \Delta N^{-}-M \bowtie \Delta N^{-}\right)\right) \\
& =M \bowtie N-\left(\left(M \bowtie \Delta N^{-} \cup \Delta M^{-} \bowtie N\right)-\left(\Delta M^{-}-M\right) \bowtie \Delta N^{-}\right) \\
& =M \bowtie N-\left(M \bowtie \Delta N^{-} \cup \Delta M^{-} \bowtie N\right) \cup\left(M \bowtie N \cap\left(\Delta M^{-}-M\right) \bowtie \Delta N^{-}\right) \\
& =M \bowtie N-\left(M \bowtie \Delta N^{-} \cup \Delta M^{-} \bowtie N\right)
\end{aligned}
$$

where the final step follows because $M$ and $\Delta M^{-}-M$ are disjoint, and so $M \bowtie N \cap$ $\left(\Delta M^{-}-M\right) \bowtie \Delta N^{-}=\varnothing$.

We now proceed as follows:

$$
\begin{aligned}
M^{\prime} \bowtie N^{\prime} & =\left(\left(M-\Delta M^{-}\right) \cup \Delta M^{+}\right) \bowtie\left(\left(N-\Delta N^{-}\right) \cup \Delta N^{+}\right) \\
& =\left(M-\Delta M^{-}\right) \bowtie\left(N-\Delta N^{-}\right) \\
& \cup\left(\left(\left(M-\Delta M^{-}\right) \cup \Delta M^{+}\right) \bowtie \Delta N^{+} \cup \Delta M^{+} \bowtie\left(\left(N-\Delta N^{-}\right) \cup \Delta N^{+}\right)\right) \\
& =\left(M-\Delta M^{-}\right) \bowtie\left(N-\Delta N^{-}\right) \cup\left(M^{\prime} \bowtie \Delta N^{+} \cup \Delta M^{+} \bowtie N^{\prime}\right) \\
& =\left(M \bowtie N-\left(M \bowtie \Delta N^{-} \cup \Delta M^{-} \bowtie N\right)\right) \cup\left(M^{\prime} \bowtie \Delta N^{+} \cup \Delta M^{+} \bowtie N^{\prime}\right) \\
& =\left(M \bowtie N \cup\left(M^{\prime} \bowtie \Delta N^{+} \cup \Delta M^{+} \bowtie N^{\prime}\right)\right)-\left(M \bowtie \Delta N^{-} \cup \Delta M^{-} \bowtie N\right)
\end{aligned}
$$

where in the last step we use the disjointness of $\Delta N^{-}$with $N^{\prime}$ and $\Delta N^{+}$, and likewise of $\Delta M^{-}$with $M^{\prime}$ and $\Delta M^{+}$, and the fact that $X, Y$ disjoint implies $X \bowtie Z$ and $Y \bowtie W$ disjoint. Clearly, the positive and negative deltas resulting in this case are disjoint. Minimality follows since $M^{\prime} \bowtie \Delta N^{+} \cap M \bowtie N=\varnothing=\Delta M^{+} \bowtie N^{\prime} \cap M \bowtie N$ by minimality of $\Delta M, \Delta N$ and the fact that $X \cap Y=\varnothing$ implies $X \bowtie Z \cap Y \bowtie Z=\varnothing$, and likewise $M \bowtie \Delta N^{-} \subseteq M \bowtie N \supseteq \Delta M^{-} \bowtie N$ by minimality and monotonicity of $\bowtie$.

## Proof of part (4)

$$
\rho_{A / B}\left(M^{\prime}\right)=\rho_{A / B}\left(\left(M \cup \Delta M^{+}\right)-\Delta M^{-}\right)
$$

$$
\begin{aligned}
& =\left(\rho_{A / B}(M) \cup \rho_{A / B}\left(\Delta M^{+}\right)\right)-\rho_{A / B}\left(\Delta M^{-}\right) \\
& =\rho_{A / B}(M) \oplus\left(\rho_{A / B}\left(\Delta M^{+}\right), \rho_{A / B}\left(\Delta M^{-}\right)\right)
\end{aligned}
$$

Clearly, $\rho_{A / B}\left(\Delta M^{+}\right)$and $\rho_{A / B}\left(\Delta M^{-}\right)$are disjoint, and minimal since $\rho_{A / B}(M) \cap$ $\rho_{A / B}\left(\Delta M^{+}\right)=\varnothing$ and $\rho_{A / B}\left(\Delta M^{-}\right) \subseteq \rho_{A / B}(M)$.

## Proof of part (5)

Lemma 70. If $N \cap O=\varnothing, N^{\prime} \subseteq N$ and $M \subseteq N$, then $M \subseteq N^{\prime} \cup O$ implies $M \subseteq N^{\prime}$.
Proof.

$$
\begin{array}{rr} 
& M \subseteq N^{\prime} \cup O \\
\Rightarrow & M \cap N \subseteq\left(N^{\prime} \cup O\right) \cap N \\
\Rightarrow M \cap N \subseteq\left(N^{\prime} \cap N\right) \cup(O \cap N) & \cap \text { monotone } \\
\Rightarrow M \cap N \subseteq N^{\prime} \cap N & \\
\Rightarrow M \cap N \subseteq N^{\prime} & N^{\prime} \subseteq N ; \cap \text { indributes over } \cap \\
\Rightarrow M \cap O=\varnothing \\
\Rightarrow M \subseteq N^{\prime} & M \subseteq N ; \cap \text { induces } \subseteq
\end{array}
$$

We want to show:

If $N \subseteq M$ and $N^{\prime} \subseteq M^{\prime}$ then $(M, \Delta M) \dot{-}(N, \Delta N)=\Delta M \ominus \Delta N$.
Proof.
$\Delta N$ minimal for $N$
$\Delta N^{-} \subseteq N$
$\Delta N^{+} \cap N=\varnothing$
$\Delta M$ minimal for $M$
$\Delta M^{-} \subseteq M$
$\Delta M^{+} \cap M=\varnothing$
$N \subseteq M$
$\Delta M^{+} \cap N=\varnothing$
$\Delta O=(M, \Delta M) \dot{-}(N, \Delta N)$
$=\left(M^{\prime}-N^{\prime}\right) \ominus(M-N)$
$=\left(\left(M^{\prime}-N^{\prime}\right)-(M-N),(M-N)-\left(M^{\prime}-N^{\prime}\right)\right)$
suppose $(\Delta N, N)$
def. minimal
def. minimal
suppose $(\Delta M, M)$
def. minimal (10)
def. minimal (11)
suppose (12)
$\cap$ monotone (11) (13)
suppose ( $\Delta O$ )
delta-correctness
lem. 17 (14)

$$
\begin{align*}
& N^{\prime} \subseteq M^{\prime} \text { suppose } \\
& \Rightarrow\left(N-\Delta N^{-}\right) \cup \Delta N^{+} \subseteq\left(M-\Delta M^{-}\right) \cup \Delta M^{+} \quad \text { lem. } 16 \\
& \Rightarrow \underline{\left(N-\Delta N^{-}\right)} \subseteq \underline{\left(M-\Delta M^{-}\right)} \cup \underline{\Delta M^{+}} \quad \cdot \cup \Delta N^{+} \text {incr.; trans. } \\
& \Rightarrow \underline{N-} \underline{\Delta N^{-}} \subseteq \underline{M-\Delta M^{-}} \\
& \Rightarrow \underline{N} \subseteq\left(M-\Delta M^{-}\right) \cup \Delta N^{-} \\
& \Rightarrow N \cap \Delta M^{-} \subseteq\left(\left(M-\Delta M^{-}\right) \cup \Delta N^{-}\right) \cap \Delta M^{-} \quad \cap \text { monotone } \\
& \Rightarrow N \cap \Delta M^{-} \subseteq\left(\left(\underline{M}-\Delta M^{-}\right) \cap \Delta M^{-}\right) \cup\left(\Delta N^{-} \cap \Delta M^{-}\right) \\
& \Rightarrow N \cap \Delta M^{-} \subseteq \Delta N^{-} \cap \Delta M^{-} \\
& \Delta N^{-} \cap \Delta M^{-} \subseteq N \cap \Delta M^{-} \\
& \Rightarrow \Delta N^{-} \cap \Delta M^{-}=N \cap \Delta M^{-} \\
& \text {lem. } 701211 \\
& \text { lem. (1) } \\
& \text { distr. } \\
& \text { lem. 6. simpl. } \varnothing \text { (16) } \\
& \cap \text { monotone } 8 \\
& \text { (16); antisym. (17) } \\
& \text { 15; } \cup \text { LUB; trans. (18) } \\
& \text {. }-\Delta N^{-} \text {decr.; 12; trans. } \\
& \cap \text { induces } \subseteq  \tag{19}\\
& \cap \text { induces } \subseteq 10  \tag{20}\\
& \Delta O^{+} \\
& =\left(\underline{M^{\prime}}-\underline{N^{\prime}}\right)-\underline{(M-N)}  \tag{14}\\
& =\underline{M^{\prime}}-\left(\underline{N^{\prime}} \cup(M-N)\right) \\
& \text { lem. } 2 \\
& \left.=\left(\underline{\left(M-\Delta M^{-}\right)} \cup \underline{\Delta M^{+}}\right)-\underline{\left(\left(N-\Delta N^{-}\right) \cup \Delta N^{+} \cup(M-N)\right.}\right) \\
& \text { lem. } 16 \\
& =\left(\left(M-\Delta M^{-}\right)-\left(\left(N-\Delta N^{-}\right) \cup \Delta N^{+} \cup(M-N)\right)\right) \cup \\
& \left(\Delta M^{+}-\left(\underline{\left(N-\Delta N^{-}\right)} \cup \Delta N^{+} \cup \underline{(M-N)}\right)\right) \quad-/ \cup \text { distr } . \\
& =\left(\left(\underline{M-\Delta M^{-}}\right)-\left(\left(\underline{N-\Delta N^{-}}\right) \cup \Delta N^{+} \cup \underline{(M-N)}\right)\right) \cup\left(\Delta M^{+}-\Delta N^{+}\right) \\
& =\left(\left(\left(\underline{M}-\underline{\Delta M^{-}}\right)-(M-\underline{N})\right)-\left(\left(N-\Delta N^{-}\right) \cup \Delta N^{+}\right)\right) \cup \\
& \left(\Delta M^{+}-\Delta N^{+}\right) \quad \cup \text { comm.; lem. } 2 \\
& =\left(\underline{\left(N-\Delta M^{-}\right)}-\left(\underline{\left(N-\Delta N^{-}\right)} \cup \underline{\Delta N^{+}}\right)\right) \cup\left(\Delta M^{+}-\Delta N^{+}\right) \quad \text { lem. 33. } N \subseteq M \\
& =\left(\left(\left(\underline{N}-\underline{\Delta M^{-}}\right)-\left(N-\underline{\Delta N^{-}}\right)\right)-\Delta N^{+}\right) \cup\left(\Delta M^{+}-\Delta N^{+}\right) \\
& =\left(\left(\Delta N^{-}-\Delta M^{-}\right)-\underline{\Delta N^{+}}\right) \cup\left(\Delta M^{+}-\Delta N^{+}\right) \\
& \text {lem. } 3 \text {. } \Delta N^{-} \subseteq N \\
& =\left(\Delta N^{-}-\Delta M^{-}\right) \cup\left(\Delta M^{+}-\Delta N^{+}\right)  \tag{21}\\
& \Delta O^{-} \\
& =(M-N)-\left(M^{\prime}-N^{\prime}\right) \\
& \text { (14) } \\
& =(M-N)-\left(\left(\left(\underline{\left.M-\Delta M^{-}\right)} \cup \underline{\Delta M^{+}}\right)-\underline{N^{\prime}}\right)\right. \\
& =(M-N)-\left(\left(\left(\underline{M}-\underline{\Delta M^{-}}\right)-\underline{\left(N^{\prime}\right)}\right) \cup\left(\Delta M^{+}-N^{\prime}\right)\right) \\
& =(M-N)-\left(\left(M-\left(N^{\prime} \cup \Delta M^{-}\right)\right) \cup\left(\Delta M^{+}-(\underline{N} \oplus \underline{\Delta N})\right)\right) \\
& =(M-N)- \\
& \left(\left(M-\left(N^{\prime} \cup \Delta M^{-}\right)\right) \cup\left(\underline{\Delta M^{+}}-\left(\underline{\left(N-\Delta N^{-}\right)} \cup \underline{\Delta N^{+}}\right)\right)\right) \\
& \text {lem. } 16
\end{align*}
$$

$$
\begin{aligned}
& =(\underline{M}-\underline{N})-\underline{\left(\left(M-\left(N^{\prime} \cup \Delta M^{-}\right)\right) \cup\left(\Delta M^{+}-\Delta N^{+}\right)\right)} \\
& =\underline{M}-\left(\left(M-\left(N^{\prime} \cup \Delta M^{-}\right)\right) \cup \underline{\left.\left(\Delta M^{+}-\Delta N^{+}\right) \cup N\right)} \quad \text { lem. } 2\right. \\
& =\left(\underline{M}-\left(M-\underline{\left(N^{\prime} \cup \Delta M^{-}\right)}\right)\right)-\underline{\left(\left(\Delta M^{+}-\Delta N^{+}\right) \cup N\right)} \quad \text { lem. } 2 \\
& =\left(\underline{M} \cap\left(N^{\prime} \cup \underline{\Delta M^{-}}\right)\right)-\left(\left(\Delta M^{+}-\Delta N^{+}\right) \cup N\right) \quad \cap \text { in terms of }- \\
& =\left(\underline{M} \cap\left(\left(N-\Delta N^{-}\right) \cup \underline{\Delta N^{+}} \cup \Delta M^{-}\right)\right)-\left(\left(\Delta M^{+}-\Delta N^{+}\right) \cup N\right) \quad \text { lem. } 16 \\
& =\left(\left(N-\Delta N^{-}\right) \cup\left(M \cap \Delta N^{+}\right) \cup \Delta M^{-}\right)-\left(\left(\Delta M^{+}-\Delta N^{+}\right) \cup N\right) \quad \text { distr.; } 1920 \\
& =\left(\left(M \cap \Delta N^{+}\right) \cup \Delta M^{-}\right)-\left(\left(\Delta M^{+}-\Delta N^{+}\right) \cup N\right) \quad \text { lem. } 8 \\
& =\left(\underline{\left(M \cap \Delta N^{+}\right)}-\left(\underline{\left(\Delta M^{+}-\Delta N^{+}\right)} \cup \underline{N}\right)\right) \cup \\
& \left(\underline{\Delta M^{-}}-\left(\left(\Delta M^{+}-\Delta N^{+}\right) \cup \underline{N}\right)\right) \quad-/ \cup \text { distr. } \\
& =\left(\underline{M} \cap \underline{\Delta N^{+}}\right) \cup\left(\Delta M^{-}-N\right) \\
& =\left(\Delta N^{+}-\Delta M^{+}\right) \cup\left(\Delta M^{-}-N\right) \\
& =\left(\Delta N^{+}-\Delta M^{+}\right) \cup\left(\Delta M^{-}-\Delta N^{-}\right) \\
& \Delta O=\left(\Delta M^{+}, \Delta M^{-}\right) \oplus\left(\Delta N^{-}, \Delta N^{+}\right) \\
& =\Delta M \ominus \Delta N \\
& \text { (13); lem. } 9 \\
& \text { lem. } 2 \\
& \text { lem. } 2 \\
& \text { 111; lem. } 9 \\
& \text { lem. } 7 \text { (18) } \\
& \text { lem. } 5 \text { (27) } \\
& \text { def. } \cdot \oplus \cdot 21,22 \\
& \text { def. } \cdot \ominus \text {. }
\end{aligned}
$$

## Join Alternative

This is an alternative but equivalent definition of the join delta used in the implementation but not used in the formulation in Chapter 3. As it is not essential, the proofs are presented are less formal.

The definition requires the following definition for the join operator between sets and deltas:

Definition 29. Given the sets $M$ and the delta $\Delta N$, define the join between the two as:

$$
M \bowtie \Delta N=\left(M \bowtie \Delta N^{+}, M \bowtie \Delta N^{-}\right)
$$

The delta join operation can then be defined as follows. Note that it is not as efficient as the other definition, as more calculations are required

Definition 30. Given the sets $M, N$ and minimal deltas $\Delta M, \Delta N$, define $\dot{\bowtie}$ as follows:

$$
(M, \Delta M) \dot{\bowtie}(N, \Delta N)=((M \oplus \Delta M) \bowtie \Delta N) \oplus(\Delta M \bowtie(N \oplus \Delta N))
$$

## Lemma 71.

$$
\begin{aligned}
& ((M \oplus \Delta M) \bowtie \Delta N) \oplus(\Delta M \bowtie(N \oplus \Delta N)) \\
& \quad=\left((M \oplus \Delta M) \bowtie \Delta N^{+} \cup \Delta M^{+} \bowtie(N \oplus \Delta N),(M \oplus \Delta M) \bowtie \Delta N^{-} \cup(N \oplus \Delta N) \bowtie \Delta M^{-}\right)
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& ((M \oplus \Delta M) \bowtie \Delta N) \oplus(\Delta M \bowtie(N \oplus \Delta N)) \\
& =\left((M \oplus \Delta M) \bowtie \Delta N^{+},(M \oplus \Delta M) \bowtie \Delta N^{-}\right) \\
& \quad\left(\left(\left(\left(M \oplus M^{+} \bowtie(N \oplus \Delta N), \Delta M^{-} \bowtie(N \oplus \Delta N)\right)\right.\right.\right. \\
& \quad \cup\left(\Delta M^{+} \bowtie(N \oplus \Delta N)-(M \oplus \Delta M) \bowtie \Delta N^{-}\right), \\
& \quad\left((M \oplus \Delta M) \bowtie \Delta N^{-}-\Delta M^{+} \bowtie(N \oplus \Delta N)\right) \\
& \quad \cup\left((N \oplus \Delta N) \bowtie \Delta M^{-} \bowtie \Delta M^{+} \bowtie \bowtie(M \oplus \Delta N)\right) \\
& \quad(N M)))
\end{aligned}
$$

$(M \oplus \Delta M)$ is disjoint from $\Delta M^{-}$
it follows that:
$(M \oplus \Delta M) \bowtie \Delta N^{+}$is disjoint from $\Delta M^{-} \bowtie(N \oplus \Delta N)$
$\Longrightarrow(M \oplus \Delta M) \bowtie \Delta N^{+}-\Delta M^{-} \bowtie(N \oplus \Delta N)=(M \oplus \Delta M) \bowtie \Delta N^{+}$
similarly applies to other terms yielding:

$$
\begin{aligned}
& ((M \oplus \Delta M) \bowtie \Delta N) \oplus(\Delta M \bowtie(N \oplus \Delta N)) \\
& =\left((M \oplus \Delta M) \bowtie \Delta N^{+} \cup \Delta M^{+} \bowtie(N \oplus \Delta N),\right. \\
& \left.\quad(M \oplus \Delta M) \bowtie \Delta N^{-} \cup(N \oplus \Delta N) \bowtie \Delta M^{-}\right)
\end{aligned}
$$

## Proof of Lemma 27

To prove this property, we first observe that $\operatorname{revise}_{F}(M, N)$ can be written as $\operatorname{map}_{f}(M)=$ $\{f(m) \mid m \in M\}$ where $f=\operatorname{recrevise}_{F}(\cdot, N)$.

Lemma 72. Suppose $\Delta M$ is minimal with respect to $M$. Then

$$
\delta \operatorname{map}_{f}(M, \Delta M)=\left(\operatorname{map}_{f}\left(\Delta M^{+}\right)-\operatorname{map}_{f}(M), \operatorname{map}_{f}\left(\Delta M^{-}\right)-\operatorname{map}_{f}(M \oplus \Delta M)\right)
$$

Proof. We first observe that

$$
\begin{align*}
& \operatorname{map}_{f}(M \cup N)=\operatorname{map}_{f}(M) \cup\left(\operatorname{map}_{f}(N)-\operatorname{map}_{f}(M)\right)  \tag{23}\\
& \operatorname{map}_{f}(M-N)=\operatorname{map}_{f}(M)-\left(\operatorname{map}_{f}(N)-\operatorname{map}_{f}(M-N)\right)
\end{align*}
$$

These equations are easy to show by calculation. Combining them we have

$$
\begin{aligned}
& \operatorname{map}_{f}(M \oplus \Delta M) \\
& \quad=\operatorname{map}_{f}\left(\left(M \cup \Delta M^{+}\right)-\Delta M^{-}\right) \\
& \quad=\operatorname{map}_{f}\left(M \cup \Delta M^{+}\right)-\left(\operatorname{map}_{f}\left(\Delta M^{-}\right)-\operatorname{map}_{f}\left(\left(M \cup \Delta M^{+}\right)-\Delta M^{-}\right)\right) \\
& \quad=\left(\operatorname{map}_{f}(M) \cup\left(\operatorname{map}_{f}\left(\Delta M^{+}\right)-\operatorname{map}_{f}(M)\right)\right)-\left(\operatorname{map}_{f}\left(\Delta M^{-}\right)-\operatorname{map}_{f}(M \oplus \Delta M)\right) \\
& \quad=\operatorname{map}_{f}(M) \oplus\left(\operatorname{map}_{f}\left(\Delta M^{+}\right)-\operatorname{map}_{f}(M), \operatorname{map}_{f}\left(\Delta M^{-}\right)-\operatorname{map}_{f}(M \oplus \Delta M)\right)
\end{aligned}
$$

Moreover, it is easy to see that $\operatorname{map}_{f}\left(\Delta M^{+}\right)-\operatorname{map}_{f}(M)$ is disjoint from $\operatorname{map}_{f}(M)$ and $\operatorname{map}_{f}\left(\Delta M^{-}\right)-\operatorname{map}_{f}(M \oplus \Delta M) \subseteq \operatorname{map}_{f}(M)$, assuming $\Delta M$ is minimal with respect to $M$. Hence by uniqueness of minimal deltas, $\operatorname{\delta map}_{f}(M, \Delta M)=\left(\operatorname{map}_{f}\left(\Delta M^{+}\right)-\right.$ $\left.\operatorname{map}_{f}(M), \operatorname{map}_{f}\left(\Delta M^{-}\right)-\operatorname{map}_{f}(M \oplus \Delta M)\right)$ as required.

Lemma 73. If $f$ is injective on $M \cup N$ and $M, N$ are disjoint then so ${\operatorname{are~} \operatorname{map}_{f}(M)}$ and $\operatorname{map}_{f}(N)$.

Proof. Suppose $x \in \operatorname{map}_{f}(M) \cap \operatorname{map}_{f}(N)$. Then $x=f(y)=f(z)$ for some $y \in M$ and $z \in N$. By injectivity, $f(y)=f(z)$ implies $y=z$, which is impossible since $M$ and $N$ are disjoint.

Lemma 74. If $f$ is injective on $M$ and on $M \oplus \Delta M$, where $\Delta M$ is minimal for $M$, then

$$
\delta \operatorname{map}_{f}(M, \Delta M)=\left(\operatorname{map}_{f}\left(\Delta M^{+}\right), \operatorname{map}_{f}\left(\Delta M^{-}\right)\right)
$$

Proof. By Lem. 72 we have

$$
\delta \operatorname{map}_{f}(M, \Delta M)=\left(\operatorname{map}_{f}\left(\Delta M^{+}\right)-\operatorname{map}_{f}(M), \operatorname{map}_{f}\left(\Delta M^{-}\right)-\operatorname{map}_{f}(M \oplus \Delta M)\right)
$$

Using Lem. 73 since $f$ is injective on $M$, and $\Delta M$ is minimal, we have

$$
\operatorname{map}_{f}\left(\Delta M^{+}\right) \cap \operatorname{map}_{f}(M)=\varnothing
$$

hence, $\operatorname{map}_{f}\left(\Delta M^{+}\right)-\operatorname{map}_{f}(M)=\operatorname{map}_{f}\left(\Delta M^{+}\right)$. Using the same lemma since $f$ is injective on $M \oplus \Delta M$, since minimality implies $(M \oplus \Delta M) \cap \Delta M^{-}=\varnothing$, we have

$$
\operatorname{map}_{f}\left(\Delta M^{-}\right) \cap \operatorname{map}_{f}(M \oplus \Delta M)=\varnothing
$$

and so $\operatorname{map}_{f}\left(\Delta M^{-}\right)-\operatorname{map}_{f}(M \oplus \Delta M)=\operatorname{map}_{f}\left(\Delta M^{-}\right)$. The desired result follows.
Lemma 75. recrevise ${ }_{X \rightarrow A}(m, N) \backslash_{A}=m \backslash_{A}$.
Proof. There are two cases according to the definition of recrevise ${ }_{X \rightarrow A}(m, N)$. If $m$ is unchanged the result is immediate, otherwise the result is $m \leftarrow n[A]$ for some $n \in N$ where $n[X]=m[X]$, and only the $A$ field is changed.

Lemma 76. If $M \vDash X \rightarrow A$ then recrevise $X_{X \rightarrow A}(\cdot, N)$ is injective on $M$; that is, for $m, m^{\prime} \in M$ we have

$$
\text { recrevise }_{X \rightarrow A}(m, N)=\text { recrevise }_{X \rightarrow A}\left(m^{\prime}, N\right) \Longrightarrow m=m^{\prime}
$$

Proof. Clearly $A \notin X$ since $\{X \rightarrow A\}$ is required to be in tree form. Let $m, m^{\prime} \in M$ be given and assume that recrevise ${ }_{X \rightarrow A}(m, N)=$ recrevise $_{X \rightarrow A}\left(m^{\prime}, N\right)$. By Lem. 75, we know also that

$$
m \backslash_{A}=\text { recrevise }_{X \rightarrow A}(m, N) \backslash_{A}=\text { recrevise }_{X \rightarrow A}\left(m^{\prime}, N\right) \backslash_{A}=m^{\prime} \backslash_{A}
$$

Since $A \notin X$, it follows that $m[X]=m^{\prime}[X]$, so $m[A]=m^{\prime}[A]$ because $M \vDash X \rightarrow A$. Thus,

$$
m=m \backslash_{A} \leftarrow m[A]=m^{\prime} \backslash_{A} \leftarrow m^{\prime}[A]=m^{\prime}
$$

Hence, recrevise ${ }_{X \rightarrow A}(\cdot, N)$ is injective.

Lemma 27. Suppose $M \models X \rightarrow A$ and $M \oplus \Delta M \models X \rightarrow A$. Then

$$
\text { drevise }_{X \rightarrow A}((M, \Delta M),(N, \varnothing))=\left(\text { revise }_{X \rightarrow A}\left(\Delta M^{+}, N\right) \text {,revise }_{X \rightarrow A}\left(\Delta M^{-}, N\right)\right) .
$$

Proof. Let $f=$ recrevise $_{X \rightarrow A}(\cdot, N)$. Then by Lem. 76, $f$ is injective on $M$ and $M \oplus \Delta M$, so we have:

$$
\begin{align*}
\text { revise }_{X \rightarrow A}(M \oplus \Delta M, N) & =\operatorname{map}_{f}(M \oplus \Delta M)  \tag{24}\\
& =\operatorname{map}_{f}(M) \oplus \delta \operatorname{map}_{f}(M, \Delta M)  \tag{25}\\
& =\operatorname{revise}_{X \rightarrow A}(M, N) \oplus \operatorname{map}_{f}(M, \Delta M)
\end{align*}
$$

which implies

$$
\begin{align*}
\text { drevise }_{X \rightarrow A}((M, \Delta M),(N, \varnothing)) & =\operatorname{jop}_{f}(M, \Delta M)  \tag{26}\\
& =\left(\operatorname{map}_{f}\left(\Delta M^{+}\right), \operatorname{map}_{f}\left(\Delta M^{-}\right)\right)  \tag{27}\\
& =\left(\operatorname{revise}_{X \rightarrow A}\left(\Delta M^{+}, N\right), \text { revise }_{X \rightarrow A}\left(\Delta M^{-}, N\right)\right)
\end{align*}
$$

by uniqueness of minimal deltas.

## Proof of Lemma 28

Lemma 77. Given $m, N, N^{\prime}: U$ with $X \subseteq U$, then exactly one of the following holds:

- There is no $n \in N^{\prime}$ with $m[X]=n[X]$
- There exists $n \in N^{\prime}-N$ with $m[X]=n[X]$.
- There exists $n \in N^{\prime} \cap N$ with $m[X]=n[X]$, but no $n^{\prime} \in N^{\prime}-N$ with $m[X]=n^{\prime}[X]$.

Proof. It is easy to see that the three cases are mutually exclusive. To see that they are exhaustive, suppose the first and second cases do not hold. The failure of the first case implies that there is an $n \in N^{\prime}$ with $m[X]=n[X]$ and the failure of the second case implies that such an $n$ must fall in $N^{\prime} \cap N$ and there can be no other $n^{\prime} \in N^{\prime}-N$ satisfying $m[X]=n^{\prime}[X]$, as required.

Definition 31. Suppose $m, N, N^{\prime}, F: U$ are given. For the sake of an inductive invariant for lem. 779, we define

$$
\begin{aligned}
& \operatorname{Inv}_{F}\left(m, N, N^{\prime}\right) \Longleftrightarrow \forall X \rightarrow Y \in F \text {. if } \nexists n \in\left(N^{\prime}-N\right) . m[X]=n[X] \\
& \quad \text { then } m,\left(N^{\prime} \cap N\right) \vDash X \rightarrow Y
\end{aligned}
$$

Lemma 78. Given $m, N, N^{\prime}, F: U$, we have:

1. $m \in N \models F$ implies $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$
2. $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$ implies $\operatorname{Inv}_{F^{\prime}}\left(m, N, N^{\prime}\right)$ whenever $F^{\prime} \subseteq F$.
3. If $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$ and $F=\{X \rightarrow Y\} \cdot F^{\prime}$ and there exists $n \in N^{\prime}-N$ with $n[X]=$ $m[X]$ then $\operatorname{Inv}_{F^{\prime}}\left(m \leftarrow n[Y], N, N^{\prime}\right)$.

Proof. 1. Since $m, N \vDash F$ we know that $m, N \vDash X \rightarrow Y$ for all $X \rightarrow Y \in F$. Clearly also $m,\left(N \cap N^{\prime}\right) \vDash X \rightarrow Y$.
2. For the second part, assume $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$. Suppose $X \rightarrow Y \in F^{\prime} \subseteq F$ is given. Then $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$ implies that if $\nexists n \in\left(N^{\prime}-N\right) \cdot m[X]=n[X]$ then $m,\left(N^{\prime} \cap\right.$ $N) \vDash X \rightarrow Y$, as required.
3. Suppose $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$ and $F=\{X \rightarrow Y\} \cdot F^{\prime}$ and there exists $n \in N^{\prime}-N$ with $n[X]=m[X]$. To show $\operatorname{Inv}_{F^{\prime}}\left(m \leftarrow n[Y], N, N^{\prime}\right)$, let $X^{\prime} \rightarrow Y^{\prime} \in F^{\prime}$ be given. We must show that if $\nexists n^{\prime} \in N^{\prime}-N$ with $n^{\prime}\left[X^{\prime}\right]=(m \leftarrow n[Y])\left[X^{\prime}\right]$ then $m \leftarrow n[Y],(N \cap$ $\left.N^{\prime}\right) \models X^{\prime} \rightarrow Y^{\prime}$.

Because $F$ is in tree form, we know that $X^{\prime}, Y^{\prime}, X, Y$ are all either disjoint or identical, and $X \neq Y, X^{\prime} \neq Y^{\prime}$. We consider the following cases:

- $Y^{\prime}=X$ is impossible since $X$ was a root and $X^{\prime} \rightarrow X \in F^{\prime}$ implies that $X$ could not be a root.
- $Y^{\prime}=Y$ is impossible since there cannot be two FDs $X \rightarrow Y, X^{\prime} \rightarrow Y$ in $F$ since it is in tree form.
- $X^{\prime}=X$ is impossible since by assumption $n \in N^{\prime}-N$ satisfies $n[X]=$ $m[X]=(m \leftarrow n[Y])[X]$, since $X, Y$ are disjoint, which contradicts the assumption that $\nexists n^{\prime} \in N^{\prime}-N$ with $n^{\prime}[X]=(m \leftarrow n[Y])[X]$.
- $X^{\prime}=Y$ is impossible since $n[Y]=(m \leftarrow n[Y])[Y]$, which contradicts the assumption that $\nexists n^{\prime} \in N^{\prime}-N$ with $n^{\prime}[Y]=(m \leftarrow n[Y])[Y]$.
- If $X^{\prime}$ and $Y^{\prime}$ are both disjoint from $X$ and $Y$ then using $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$ we can conclude $m,\left(N \cap N^{\prime}\right) \models X^{\prime} \rightarrow Y^{\prime}$, and since $X^{\prime}, Y^{\prime}$ are disjoint from $Y$ we can also conclude $m \leftarrow n[Y],\left(N \cap N^{\prime}\right) \models X^{\prime} \rightarrow Y^{\prime}$ since updating $m[Y]$ to $n[Y]$ has no effect on the values of $m\left[X^{\prime}\right]$ or $m\left[Y^{\prime}\right]$.

Lemma 79. Suppose $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$. Then

$$
\operatorname{recrevise}_{F}\left(m, N^{\prime}\right)=\operatorname{recrevise}_{F}\left(m, N^{\prime}-N\right) .
$$

Proof. Let $N, N^{\prime}$ be given. We prove by induction on $F$ that for any $m$, if $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$ then recrevise ${ }_{F}\left(m, N^{\prime}\right)=\operatorname{recrevise}_{F}\left(m, N^{\prime}-N\right)$.

- If $F=\varnothing$ the desired conclusion is immediate.
- If $F=\{X \rightarrow Y\} \cdot F^{\prime}$, then suppose $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$, and note that $\operatorname{Inv}_{F^{\prime}}\left(m, N, N^{\prime}\right)$ holds by Lem. 78(2). We consider the following three subcases:
- If there is no $n \in N^{\prime}$ such that $n[X]=m[X]$, then

$$
\begin{align*}
\operatorname{recrevise}_{F}\left(m, N^{\prime}\right) & =\operatorname{recrevise}_{F^{\prime}}\left(m, N^{\prime}\right)  \tag{28}\\
& =\operatorname{recrevise}_{F^{\prime}}\left(m, N^{\prime}-N\right)  \tag{29}\\
& =\operatorname{recrevise}_{F}\left(m, N^{\prime}-N\right)
\end{align*}
$$

using the induction hypothesis on $m$ and since $N^{\prime} \supseteq N^{\prime}-N$.

- If there exists $n \in N^{\prime}-N$ such that $n[X]=m[X]$, then observe that by Lemma $78(3)$ we have $\operatorname{Inv}_{F^{\prime}}\left(m \leftarrow n[Y], N, N^{\prime}\right)$, so the induction hypothesis is available for $m \longleftarrow n[Y]$. We reason as follows:

$$
\begin{align*}
\operatorname{recrevise}_{F}\left(m, N^{\prime}\right) & =\operatorname{recrevise}_{F^{\prime}}\left(m \leftarrow n[Y], N^{\prime}\right)  \tag{30}\\
& =\operatorname{recrevise}_{F^{\prime}}\left(m \leftarrow n[Y], N^{\prime}-N\right)  \tag{31}\\
& =\operatorname{recrevise}_{F}\left(m, N^{\prime}-N\right)
\end{align*}
$$

where we use the induction hypothesis on $m \longleftarrow n[Y]$ and the fact that $n \in N^{\prime}-N$.

- If there exists $n \in N \cap N^{\prime}$ such that $n[X]=m[X]$ but no $n^{\prime} \in N^{\prime}-N$ with $n^{\prime}[X]=m[X]$ then $\operatorname{Inv}_{F}\left(m, N, N^{\prime}\right)$ implies that $m,\left(N \cap N^{\prime}\right) \vDash X \rightarrow Y$, so $m[X]=n[X]$ implies $m[Y]=n[Y]$. Moreover, $m \leftarrow n[Y]=m \leftarrow m[Y]=m$. So, we can reason as follows:

$$
\begin{align*}
\text { recrevise }_{F}\left(m, N^{\prime}\right) & =\operatorname{recrevise}_{F^{\prime}}\left(m \leftarrow n[Y], N^{\prime}\right)  \tag{32}\\
& =\operatorname{recrevise}_{F^{\prime}}\left(m, N^{\prime}\right)  \tag{33}\\
& =\operatorname{recrevise}_{F^{\prime}}\left(m, N^{\prime}-N\right)  \tag{34}\\
& =\operatorname{recrevise}_{F}\left(m, N^{\prime}-N\right)
\end{align*}
$$

using the induction hypothesis on $m$ and the fact that there was no $n^{\prime} \in$ $N^{\prime}-N$ matching $m$ on $X$.

These three cases are exhaustive by lemma 77, so the proof of the induction step is complete.

Lemma 80. Suppose $m \in N$ and $N \vDash F$. Then

$$
\operatorname{recrevise}_{F}(m, N \oplus \Delta N)=\operatorname{recrevise}_{F}\left(m, \Delta N^{+}\right) .
$$

Proof. Note that $(N \oplus \Delta N)-N=\Delta N^{+}$, and $m \in N$ implies $\operatorname{Inv}_{F}(m, N, N \oplus \Delta N)$, so by Lemma 79 we can conclude recrevise $F(m, N \oplus \Delta N)=\operatorname{recrevise}_{F}\left(m, \Delta N^{+}\right)$.

Lemma 81. If $\operatorname{merge}_{F}(M, N)=M$ and $N \oplus \Delta N \vDash F$, then $N-\Delta N^{-} \subseteq \operatorname{merge}_{F}\left(M, \Delta N^{+}\right)$.

Proof.
$\operatorname{merge}_{F}(M, N)=M \quad$ assumption
$N \subseteq M$ def. merge. $(\cdot, \cdot)$
$N-\Delta N^{-} \subseteq N \subseteq M$ . $-\Delta N^{-}$decr.; transitivity
$N \oplus \Delta N \vDash F \quad$ assumption
$m \in N-\Delta N^{-} \quad$ suppose ( $m$ ) (2)
$m \in N$

Now show recrevise $F_{F}\left(m, \Delta N^{+}\right)=m$ by induction on cardinality of $F$.
Case $F=\varnothing$ :
$\hookrightarrow \operatorname{recrevise}_{F}\left(m, \Delta N^{+}\right)=m \quad$ def. recrevise ${ }_{F}(\cdot, \cdot)$
Case $F=X \rightarrow Y \cdot F^{\prime}$ :
$N \oplus \Delta N \vDash F^{\prime}$
$N \oplus \Delta N \vDash F$ and $F^{\prime} \subseteq F$
Case $\exists n \in \Delta N^{+} . n[X]=m[X]:$

$$
\begin{array}{lr}
n[Y]=m[Y] & N \oplus \Delta N \vDash F \\
\text { recrevise }_{F}\left(m, \Delta N^{+}\right) & \\
\quad=\text { recrevise }_{F^{\prime}}\left(m \leftarrow n[Y], \Delta N^{+}\right) & \text {def. } \text { recrevise }_{F}(\cdot, \cdot) \\
\hookrightarrow \text { recrevise }_{F}\left(m, \Delta N^{+}\right) & \\
\quad=\text { recrevise }_{F^{\prime}}\left(m, \Delta N^{+}\right)=m & n[Y]=m[Y] ; \mathrm{IH}
\end{array}
$$

Case $\nexists n \in \Delta N^{+} . n[X]=m[X]:$

$$
\hookrightarrow \text { recrevise }_{F}\left(m, \Delta N^{+}\right)
$$

$$
=\text { recrevise }_{F^{\prime}}\left(m, \Delta N^{+}\right)=m
$$

def. recrevise ${ }_{F}(\cdot, \cdot)$; IH (4)
def. merge $_{F}(\cdot, \cdot)$

$$
N-\Delta N^{-}=\operatorname{revise}_{F}\left(N-\Delta N^{-}, \Delta N^{+}\right) \quad \forall \text { intro (22; def. recrevise }{ }_{F}(\cdot, \cdot)
$$

$$
\subseteq \operatorname{revise}_{F}\left(M, \Delta N^{+}\right) \quad \quad \text { recrevise }_{F}\left(\cdot, \Delta N^{+}\right) \text {monotone }
$$

$$
\subseteq \text { merge }_{F}\left(M, \Delta N^{+}\right)
$$

Lemma 28. If $\operatorname{merge}_{F}(M, N)=M$ then

$$
{\delta \operatorname{merge}_{F}((M, \varnothing),(N, \Delta N))=\operatorname{merge}_{F}\left(M, \Delta N^{+}\right) \ominus M . . . . ~}_{\text {. }}
$$

Proof.

$$
\begin{array}{rlr}
\operatorname{revise}_{F}(M, N \oplus \Delta N) & =\left\{\operatorname{recrevise}_{F}(m, N \oplus \Delta N) \mid m \in M\right\} & \text { def. revise }(\cdot, \cdot) \\
& =\left\{\operatorname{recrevise}_{F}\left(m, \Delta N^{+}\right) \mid m \in M\right\} & \text { lemma } 80 \\
& =\operatorname{revise}_{F}\left(M, \Delta N^{+}\right) & \text {def. } \operatorname{revise}(\cdot, \cdot)
\end{array}
$$

$$
\begin{align*}
& \operatorname{merge}_{F}(M, N \oplus \Delta N) \\
& \quad=\operatorname{revise}_{F}(M, N \oplus \Delta N) \cup(N \oplus \Delta N)  \tag{11}\\
& \quad=\operatorname{revise}_{F}\left(M, \Delta N^{+}\right) \cup(N \oplus \Delta N) \\
& \quad=\operatorname{revise}_{F}\left(M, \Delta N^{+}\right) \cup\left(\left(N-\Delta N^{-}\right) \cup \Delta N^{+}\right) \\
& \quad=\operatorname{revise}_{F}\left(M, \Delta N^{+}\right) \cup \Delta N^{+} \cup\left(N-\Delta N^{-}\right) \\
& \quad=\operatorname{merge}_{F}\left(M, \Delta N^{+}\right) \cup\left(N-\Delta N^{-}\right) \\
& \quad=\operatorname{merge}_{F}\left(M, \Delta N^{+}\right)
\end{align*}
$$

$$
=\operatorname{revise}_{F}(M, N \oplus \Delta N) \cup(N \oplus \Delta N) \quad \text { def. merge }(\cdot, \cdot)
$$

def. $\oplus$
comm. U; assoc. $\cup$
def. merge( $\cdot, \cdot$ )
lemma 81 (2)

```
\(\delta \operatorname{merge}_{F}((M, \varnothing),(N, \Delta N))\)
    \(=\operatorname{merge}_{F}(M, N \oplus \Delta N) \ominus \operatorname{merge}_{F}(M, N)\)
    \(=\) merge \(_{F}(M, N \oplus \Delta N) \ominus M\)
    \(=\) merge \(_{F}\left(M, \Delta N^{+}\right) \ominus M\)
```

    assumption
    
## Proof of Lemma 29

Lemma 82. If $P=\operatorname{affected}_{G}(N), F \subseteq G$ and $m \in \sigma_{\neg P}(M)$ for any $M$, then recrevise $F_{F}(m, N)=$ $m$.

Proof.
$P=\operatorname{affected}_{G}(N)$
suppose ( $P$ )
$m \in \sigma_{\neg P}(M)$
suppose ( $m$ )
Now show recrevise $F_{F}(m, N)=m$ by induction on the cardinality of $F$.
Case F $=\varnothing$ :
$\hookrightarrow$ recrevise $_{F}(m, N)=m \quad$ def. recrevise ${ }_{F}(\cdot, \cdot)$
Case $F=X \rightarrow Y \cdot F^{\prime}$ :
$\begin{array}{lr}\nexists n \in N . n[X]=m[X] & m \in \sigma_{\neg P}(M) ; \text { def. } \operatorname{affected}_{G}(\cdot) \\ \hookrightarrow \text { recrevise }_{F}(m, N)=\text { recrevise }_{F^{\prime}}(m, N)=m & \text { def. recrevise }{ }_{F}(\cdot, \cdot) ; \mathrm{IH}\end{array}$

Lemma 83. If $P=\operatorname{affected}_{F}(N)$, then revise $F(M, N)=\operatorname{revise}_{F}\left(\sigma_{P}(M), N\right) \cup \sigma_{\neg P}(M)$.

Proof.
$P=\operatorname{affected}_{F}(N) \quad$ suppose $(P)$
$m \in \sigma_{\neg P}(M) \quad$ suppose $(m)$
$m=$ recrevise $_{F}(m, N)$ lemma 82
$\operatorname{revise}_{F}\left(\sigma_{\neg P}(M), N\right)=\sigma_{\neg P}(M) \quad \forall$ intro $(m)$; def. revise ${ }_{F}(\cdot, \cdot)(1)$
$M=\sigma_{P}(M) \cup \sigma_{\neg P}(M) \quad$ decompose $M$

$$
\begin{align*}
\operatorname{revise}_{F}(M, N) & =\operatorname{revise}_{F}\left(\sigma_{P}(M), N\right) \cup \operatorname{revise}_{F}\left(\sigma_{\neg P}(M), N\right) \quad \text { def. } \operatorname{revise}_{F}(\cdot, \cdot) \\
& =\operatorname{revise}_{F}\left(\sigma_{P}(M), N\right) \cup \sigma_{\neg P}(M) \tag{1}
\end{align*}
$$

Lemma 84. If $P=\operatorname{affected}_{F}(N)$, then $\operatorname{merge}_{F}(M, N)=\operatorname{merge}_{F}\left(\sigma_{P}(M), N\right) \cup \sigma_{\neg P}(M)$.

Proof.

$$
\begin{array}{rlr}
P=\operatorname{affected}_{F}(N) & \text { suppose }(P) \\
\operatorname{merge}_{F}(M, N) & =\operatorname{revise}_{F}(M, N) \cup N & \text { def. } \operatorname{merge}_{F}(\cdot, \cdot) \\
& =\left(\operatorname{revise}_{F}\left(\sigma_{P}(M), N\right) \cup \sigma_{\neg P}(M)\right) \cup N & \text { lemma } \boxed{83} \\
& =\left(\operatorname{revise}_{F}\left(\sigma_{P}(M), N\right) \cup N\right) \cup \sigma_{\neg P}(M) & \text { comm. } \cup \\
& =\operatorname{merge}_{F}\left(\sigma_{P}(M), N\right) \cup \sigma_{\neg P}(M) & \text { def. } \operatorname{merge}_{F}(\cdot, \cdot)
\end{array}
$$

Lemma 85. If $M \cap P=N \cap O=\varnothing$, then $(M \cup N) \ominus(O \cup P)=(M \ominus O) \oplus(N \ominus P)$.
Proof. First observe that because $M \cap P=N \cap O=\varnothing$, we have

$$
\begin{aligned}
(M \cup N)-(O \cup P) & =(M-(O \cup P)) \cup(N-(O \cup P)) \quad-/ \cup \text { distr. } \\
& =(M-O) \cup(N-P)
\end{aligned}
$$

Now we proceed as follows:

$$
\begin{array}{rlr}
(M \cup N) \ominus(O \cup P) & \text { lem. 17 } \\
= & ((M \cup N)-(O \cup P),(O \cup P)-(M \cup N)) & \text { observation } \\
= & ((M-O) \cup(N-P),(O-M) \cup(P-N)) & \\
= & (((M-O)-(P-N)) \cup((N-P)-(O-M)), & \\
& ((O-M)-(N-P)) \cup((P-N)-(M-O))) & M \cap P=\varnothing ; N \cap O=\varnothing \\
= & (M-O, O-M) \oplus(N-P, P-N) & \text { def. } \oplus \\
= & (M \ominus O) \oplus(N \ominus P) & \text { lem. 17 }
\end{array}
$$

Lemma 29. If $P=\operatorname{affected}_{F}\left(\Delta N^{+}\right)$and either $F \neq \varnothing$ or $\Delta N^{+} \cap M=\varnothing$ then

$$
\operatorname{merge}_{F}\left(M, \Delta N^{+}\right) \ominus M=\operatorname{merge}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \ominus \sigma_{P}(M)
$$

Proof.
$M \cap \Delta N^{+}=\varnothing$ or $F \neq \varnothing$
assumption (1)
$P=\operatorname{affected}_{F}\left(\Delta N^{+}\right)=\bigvee_{X \rightarrow Y \in F} X \in \pi_{X}\left(\Delta N^{+}\right) \quad$ suppose $(P) ; \operatorname{def.~}^{\operatorname{affected}_{F}(\cdot)}$

First show $\operatorname{merge}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \cap \sigma_{\neg P}(M)=\varnothing$ by case analysis on F .
Case $F=\varnothing$ :

$$
\begin{array}{lr}
M \cap \Delta N^{+}=\varnothing & \boxed{1}) ; F=\varnothing \\
\operatorname{merge}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) & \\
\quad=\operatorname{merge}_{F}\left(\sigma_{\perp}(M), \Delta N^{+}\right) & \text {def. } \operatorname{affected}_{F}(\cdot) \\
=\operatorname{merge}_{F}\left(\varnothing, \Delta N^{+}\right) & \sigma_{\perp}(\cdot) \\
=\operatorname{revise}_{F}\left(\varnothing, \Delta N^{+}\right) \cup \Delta N^{+}=\Delta N^{+} & \text {def. } \operatorname{merge}_{F}(\cdot, \cdot) ; \text { def. } \operatorname{revise}_{F}(\cdot, \cdot) \\
\hookrightarrow \operatorname{merge}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \cap \sigma_{\neg P}(M) & \\
\quad=\Delta N^{+} \cap \sigma_{\neg P}(M)=\varnothing & \text { (4) }
\end{array}
$$

Case $F=X \rightarrow Y \cdot F^{\prime}$ :

## Case $m \in \Delta N^{+}$:

$$
\hookrightarrow m \notin \sigma_{\neg P}(M) \quad \text { def. } \operatorname{affected}_{F}(\cdot)
$$

$$
\hookrightarrow \operatorname{merge}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \cap \sigma_{\neg P}(M)=\varnothing
$$

$$
\forall \text { intro }(m)
$$

$$
\operatorname{merge}_{F}\left(M, \Delta N^{+}\right) \ominus M
$$

$$
=\left(\operatorname{merge}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \cup \sigma_{\neg P}(M)\right)
$$

$$
\ominus\left(\sigma_{P}(M) \cup \sigma_{\neg P}(M)\right) \quad \text { lemma } 84 \text { decompose } M
$$

$$
=\left(\operatorname{merge}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \ominus \sigma_{P}(M)\right)
$$

$$
\oplus\left(\sigma_{\neg P}(M) \ominus \sigma_{\neg P}(M)\right) \quad \text { (3); lemma } 85
$$

$$
=\operatorname{merge}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \ominus \sigma_{P}(M) \quad \text { simpl. }
$$

$$
\begin{aligned}
& P=X \in \pi_{X}\left(\Delta N^{+}\right) \vee \bigvee_{X^{\prime} \rightarrow Y^{\prime} \in F^{\prime}} X^{\prime} \in \pi_{X^{\prime}}\left(\Delta N^{+}\right) \quad \text { expand } F \text { in } P \\
& \neg P=X \notin \pi_{X}\left(\Delta N^{+}\right) \wedge \bigwedge_{X^{\prime} \rightarrow Y^{\prime} \in F^{\prime}} X^{\prime} \notin \pi_{X^{\prime}}\left(\Delta N^{+}\right) \quad \text { de Morgan } \\
& m \in \text { merge }_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \\
& =\operatorname{revise}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \cup \Delta N^{+} \quad \text { suppose }(m) \text {; def. } \text { merge }_{F}(\cdot, \cdot) \\
& \text { Case } m \in \operatorname{revise}_{F}\left(\sigma_{P}(M), \Delta N^{+}\right) \text {: } \\
& \exists m^{\prime} \in \sigma_{P}(M) \cdot m=\operatorname{recrevise}_{F}\left(m^{\prime}, \Delta N^{+}\right) \text {and } m^{\prime}[X]=m[X] \quad \text { def. } \text { revise }_{F}(\cdot, \cdot) \\
& m^{\prime}[X] \in \pi_{X}\left(\Delta N^{+}\right) \\
& m[X] \in \pi_{X}\left(\Delta N^{+}\right) \quad m[X]=m^{\prime}[X] \\
& \forall n \in \sigma_{\neg P}(M) . n[X] \neq m[X] \quad n[X] \notin \pi_{X}\left(\Delta N^{+}\right) \\
& \hookrightarrow m \notin \sigma_{\neg P}(M)
\end{aligned}
$$

## B. 2 Proofs for Section 3.2

## Proof of Theorem 3]

Lemma 86. Suppose $M \vDash F$. Then revise $F_{F}\left(\sigma_{\neg P}(M), N\right) \cap \sigma_{P}(M)=\varnothing$.

Proof.
$Z=U-$ outputs $(F)$
suppose ( $Z$ )
$m \in \sigma_{\neg P}(M)$
suppose ( $m$ )
Case recrevise ${ }_{F}(m, N)=m$ :
$\hookrightarrow m \notin \sigma_{P}(M)$
$m \in \sigma_{\neg P}(M)$
Case recrevise ${ }_{F}(m, N)=m^{\prime} \neq m$ :
$m^{\prime}[Z]=m[Z] \quad$ def. recrevise ${ }_{F}(\cdot, \cdot)$
$\forall m^{\prime \prime} \in M$ and $m[Z]=m^{\prime \prime}[Z] . m=m^{\prime \prime} \neq m^{\prime}$
$\hookrightarrow m^{\prime} \notin \sigma_{P}(M)$
$M \models F$
$m^{\prime} \notin M$

Lemma 87. Suppose $M \vDash F$. Then revise $F_{F}\left(\sigma_{\neg P}(M), N\right)=\operatorname{revise}_{F}\left(\sigma_{\neg P}(M), N-\sigma_{P}(M)\right)$.
Proof. It suffices to show that if $m \in \sigma_{\neg P}(M)$, then $\operatorname{recrevise}_{F}(m, N)=\operatorname{recrevise}_{F}(m, N-$ $\sigma_{P}(M)$ ), by the definition of relational revision. This follows from Lemma 79 provided that $\operatorname{Inv}_{F}\left(m, \sigma_{P}(M), N\right)$ holds. To show this, let $X \rightarrow Y \in F$ be given and assume that there is no $n \in N-\sigma_{P}(M)$ such that $n[X]=m[X]$. We need to show that $m,\left(\sigma_{P}(M) \cap N\right) \vDash X \rightarrow Y$. This is immediate since $\{m\} \cup\left(\sigma_{P}(M) \cap N\right) \subseteq M \vDash F$.

Lemma 88. If $M \vDash F$ then $\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), N\right)=\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), N-\sigma_{P}(M)\right) \cup$ $\left(\sigma_{P}(M) \cap N\right)$.

Proof.

$$
\begin{array}{rlr} 
& \operatorname{merge}_{F}\left(\sigma_{\neg P}(M), N\right) & \\
= & \operatorname{revise}_{F}\left(\sigma_{\neg P}(M), N\right) \cup N & \text { def. } \operatorname{merge}_{F}(\cdot, \cdot) \\
= & \operatorname{revise}_{F}\left(\sigma_{\neg P}(M), N-\sigma_{P}(M)\right) \cup N & \text { lem. } 87 \\
= & \operatorname{revise}_{F}\left(\sigma_{\neg P}(M), N-\sigma_{P}(M)\right) \cup\left(N-\sigma_{P}(M)\right) \cup\left(\sigma_{P}(M) \cap N\right) & \text { decompose } N
\end{array}
$$

$$
=\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), N-\sigma_{P}(M)\right) \cup\left(\sigma_{P}(M) \cap N\right) \quad \text { def. } \operatorname{merge}_{F}(\cdot, \cdot)
$$

Lemma 89. Suppose $M \models F$ and $\Delta N$ minimal for $\sigma_{P}(M)$. Then

$$
\begin{gathered}
\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \sigma_{P}(M) \oplus \Delta N\right) \ominus \operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \sigma_{P}(M)\right) \\
\quad=\left(\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \Delta N^{+}\right) \ominus \sigma_{\neg P}(M)\right) \ominus \Delta N^{-} .
\end{gathered}
$$

Proof.

$$
\begin{align*}
& \left(\sigma_{P}(M) \oplus \Delta N\right)-\sigma_{P}(M)=\Delta N^{+} \\
& \text {lem. } 21 \text { (1) } \\
& \text { revise }_{F}\left(\sigma_{\neg P}(M), \Delta N^{+}\right) \cap \sigma_{P}(M)=\varnothing \\
& \text { (11) lem. } 86 \\
& \Delta N^{+} \cap \sigma_{P}(M)=\varnothing  \tag{1}\\
& \left(\text { revise }_{F}\left(\sigma_{\neg P}(M), \Delta N^{+}\right) \cup \Delta N^{+}\right) \cap \sigma_{P}(M)=\varnothing \\
& \operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \Delta N^{+}\right) \cap \sigma_{P}(M)=\varnothing \\
& \sigma_{P}(M) \cap \sigma_{\neg P}(M)=\varnothing \\
& \sigma_{P}(M) \cap\left(\sigma_{P}(M) \oplus \Delta N\right) \cap \sigma_{\neg P}(M)=\varnothing \quad \cap \text { monotone (3) } \\
& \left(\sigma_{P}(M) \cap\left(\sigma_{P}(M) \oplus \Delta N\right)\right) \ominus \sigma_{P}(M)=\ominus \Delta N^{-} \text {lem. 22 (4) } \\
& \text { merge }_{F}\left(\sigma_{\neg P}(M), \sigma_{P}(M) \oplus \Delta N\right) \ominus \text { merge }_{F}\left(\sigma_{\neg P}(M), \sigma_{P}(M)\right) \\
& =\left(\operatorname{merge}_{F}\left(\sigma_{\neg P}(M),\left(\sigma_{P}(M) \oplus \Delta N\right)-\sigma_{P}(M)\right)\right. \\
& \left.\cup\left(\sigma_{P}(M) \cap\left(\sigma_{P}(M) \oplus \Delta N\right)\right)\right) \\
& \ominus\left(\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \sigma_{P}(M)-\sigma_{P}(M)\right) \cup\left(\sigma_{P}(M) \cap \sigma_{P}(M)\right)\right) \quad \text { lem. } 88 \text { lem. } 88 \\
& =\left(\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \Delta N^{+}\right) \cup\left(\sigma_{P}(M) \cap\left(\sigma_{P}(M) \oplus \Delta N\right)\right)\right) \\
& \ominus\left(\text { merge }_{F}\left(\sigma_{\neg P}(M), \varnothing\right) \cup \sigma_{P}(M)\right) \quad \text { 11; simpl. } \\
& =\left(\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \Delta N^{+}\right) \cup\left(\sigma_{P}(M) \cap\left(\sigma_{P}(M) \oplus \Delta N\right)\right)\right) \\
& \ominus\left(\sigma_{\neg P}(M) \cup \sigma_{P}(M)\right) \\
& \operatorname{merge}_{F}(\cdot, \varnothing)=i d \\
& =\left(\text { merge }_{F}\left(\sigma_{\neg P}(M), \Delta N^{+}\right) \ominus \sigma_{\neg P}(M)\right) \\
& \oplus\left(\left(\sigma_{P}(M) \cap\left(\sigma_{P}(M) \oplus \Delta N\right)\right) \ominus \sigma_{P}(M)\right) \quad \text { 2. 3); lem. } 85 \\
& =\left(\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \Delta N^{+}\right) \ominus \sigma_{\neg P}(M)\right) \ominus \Delta N^{-} \quad \text { (4); def. } \ominus
\end{align*}
$$

Theorem 3. [Correctness of optimised select lens] Suppose $N=\sigma_{P}(M)$ where $M$ : $\operatorname{Rel}(U, Q, F)$. Suppose also that $\Delta N$ is minimal with respect to $N$ and that $N \oplus \Delta N$ : $\operatorname{Rel}(U, P \wedge Q, F)$. Then $\delta p u t_{\ell}(M, \Delta N)=\operatorname{dput}_{\ell^{\prime}}(M, \Delta N)$.

Proof.

$$
\begin{array}{lr}
N=\operatorname{get}_{\ell}(M) & \text { suppose }(N, M) \\
\left(M_{0}, \Delta M_{0}\right)=\operatorname{merge}_{F}^{\dagger}\left(\sigma_{\neg P}^{\dagger}(M, \varnothing),(N, \Delta N)\right) & \text { suppose }\left(M_{0}, \Delta M_{0}, \Delta N\right) \\
\left(N_{\#}, \Delta N_{\#}\right)=\sigma_{P}^{\dagger}\left(M_{0}, \Delta M_{0}\right)-^{\dagger}(N, \Delta N) & \text { suppose }\left(N_{\#}, \Delta N_{\#}\right) \\
\delta p u t_{\ell}(M, \Delta N)=\left(M_{0}, \Delta M_{0}\right) \dot{-}\left(N_{\#}, \Delta N_{\#}\right) & \text { def. } \delta p u t_{\ell}  \tag{2}\\
N=\sigma_{P}(M) & \text { def. get } t_{\ell} \\
M_{0}=\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), N\right) & \text { def. . } ;(\cdot, \cdot) \text { reflects }= \\
=\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \sigma_{P}(M)\right) & N=\sigma_{P}(M) \\
=\sigma_{\neg P}(M) \cup \sigma_{P}(M) & M \models F ; \text { defs. } \operatorname{revise}_{F}(\cdot, \cdot), \operatorname{merge}_{F}(\cdot, \cdot) \\
\quad=M &
\end{array}
$$

$N \subseteq N=\sigma_{P}(M)=\sigma_{P}\left(M_{0}\right)$

$$
\begin{array}{lr}
M_{0} \oplus \Delta M_{0}=\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), N \oplus \Delta N\right) & \text { delta-correctness } \\
N \oplus \Delta N \subseteq M_{0} \oplus \Delta M_{0} & \text { def. merge } \\
\sigma_{P}(N \oplus \Delta N) \subseteq \sigma_{P}\left(M_{0} \oplus \Delta M_{0}\right) & \text { monotone } \sigma_{P}(\cdot) \\
N \oplus \Delta N \subseteq \sigma_{P}\left(M_{0} \oplus \Delta M_{0}\right) & \text { idemp. } \sigma_{P}(\cdot) \\
\quad \subseteq \sigma_{P}\left(M_{0}\right) \oplus \dot{\sigma}_{P}\left(M_{0}, \Delta M_{0}\right) & \text { delta-correctness } \tag{4}
\end{array}
$$

$$
\left(N_{\#}, \Delta N_{\#}\right)
$$

$$
=\left(\sigma_{P}\left(M_{0}\right)-N,\left(\sigma_{P}\left(M_{0}\right), \dot{\sigma}_{P}\left(M_{0}, \Delta M_{0}\right)\right) \dot{-}(N, \Delta N)\right)
$$

$$
\Delta N_{\#}=\left(\sigma_{P}\left(M_{0}\right), \dot{\sigma}_{P}\left(M_{0}, \Delta M_{0}\right)\right) \dot{\succ}(N, \Delta N)
$$

$$
=\dot{\sigma}_{P}\left(M_{0}, \Delta M_{0}\right) \ominus \Delta N
$$

$$
=\left(\sigma_{P}\left(\Delta M_{0}^{+}\right), \sigma_{P}\left(\Delta M_{0}^{-}\right)\right) \ominus \Delta N
$$

$$
\begin{equation*}
Q=\operatorname{affected}_{F}\left(\Delta N^{+}\right) \tag{5}
\end{equation*}
$$

$$
\Delta M_{0}=\delta \operatorname{merge}_{F}\left(\left(\sigma_{\neg P}(M), \dot{\sigma}_{\neg P}(M, \varnothing)\right),(N, \Delta N)\right)
$$

$$
=\delta \operatorname{merge}_{F}\left(\left(\sigma_{\neg P}(M), \varnothing\right),(N, \Delta N)\right)
$$

$$
=\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), N \oplus \Delta N\right) \ominus \operatorname{merge}_{F}\left(\sigma_{\neg P}(M), N\right)
$$

$$
=\left(\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \Delta N^{+}\right) \ominus \sigma_{\neg P}(M)\right) \ominus \Delta N^{-}
$$

$$
=\left(\operatorname{merge}_{F}\left(\sigma_{Q \wedge \neg P}(M), \Delta N^{+}\right) \ominus \sigma_{Q \wedge \neg P}(M)\right) \ominus \Delta N^{-}
$$

$$
\begin{equation*}
\delta p u t_{\ell^{\prime}}(M, \Delta N)=\Delta M_{0} \ominus \Delta N_{\#} \tag{6}
\end{equation*}
$$

$$
\begin{equation*}
N_{\#} \subseteq N_{\#}=\sigma_{P}\left(M_{0}\right)-N \subseteq \sigma_{P}\left(M_{0}\right) \subseteq M_{0} \tag{7}
\end{equation*}
$$

def. . ${ }^{\dagger}$
$(\cdot, \cdot)$ reflects $=$ (3. 4); lem. 26 part 5 lem. 26 part 1 suppose ( $Q$ )
def. ${ }^{\dagger} ;(\cdot, \cdot)$ reflects $=$ delta-correctness def. $\delta o p$ lemma 89 (5); lemma 29 def. $\delta p u t_{\ell^{\prime}}$ def. $N_{\#}$

$$
\begin{align*}
& N_{\#} \oplus \Delta N_{\#} \\
& \quad=\sigma_{P}\left(M_{0} \oplus \Delta M_{0}\right)-(N \oplus \Delta N) \\
& \quad \subseteq \sigma_{P}\left(M_{0} \oplus \Delta M_{0}\right) \\
& \quad \subseteq M_{0} \oplus \Delta M_{0}  \tag{8}\\
& \left(M_{0}, \Delta M_{0}\right)-\left(N_{\#}, \Delta N_{\#}\right)=\Delta M_{0} \ominus \Delta N_{\#}  \tag{9}\\
& \delta_{\text {put }}(M, \Delta N)=\text { pput }_{\ell}(M, \Delta N)
\end{align*}
$$

## Proof of Theorem 4

Theorem 4. [Correctness of optimised project lens] Suppose $M: \operatorname{Rel}(U, P, F)$ and $N=\pi_{U-A}(M)$. Suppose also that $\Delta N$ is minimal with respect to $N$ and that $N \oplus \Delta N$ : $\operatorname{Rel}\left(U-A, \pi_{U-A}(P), F^{\prime}\right)$, where $F \equiv F^{\prime} \uplus\{X \rightarrow A\}$. Then $\delta p u t_{\ell}(M, \Delta N)=\delta p u t_{\ell^{\prime}}(M, \Delta N)$.

Proof.
$N=g e t_{\ell}(M)$
$\left(M^{\prime}, \Delta M^{\prime}\right)=(N, \Delta N) \bowtie^{\dagger}(\{\{A=a\}\}, \varnothing)$
suppose $(N, M)$
suppose $\left(M^{\prime}, \Delta M^{\prime}, \Delta N\right)$
$\delta \operatorname{sut}_{\ell}(M, \Delta N)=$ Srevise $_{X \rightarrow A}\left(\left(M^{\prime}, \Delta M^{\prime}\right),(M, \varnothing)\right)$
def. $\delta p u t_{\ell}$

$$
\begin{aligned}
\Delta M^{\prime} & =(N, \Delta N) \dot{\bowtie}(\{\{A=a\}\}, \varnothing) & \text { def. } .^{\dagger} ;(\cdot, \cdot) \text { reflects }= \\
& =\left(\Delta N^{+} \bowtie\{\{A=a\}\}, \Delta N^{-} \bowtie\{\{A=a\}\}\right) & \text { lemma } 26 \text { part } 3
\end{aligned}
$$

$$
\delta p u t_{\ell^{\prime}}(M, \Delta N)
$$

$$
=\left(\text { revise }_{X \rightarrow A}\left(\Delta M^{\prime+}, M\right) \text {, } \text { revise }_{X \rightarrow A}\left(\Delta M^{\prime-}, M\right)\right) \quad \text { def. } \delta \text { put }_{\ell^{\prime}}
$$

$$
\left(\text { revise }_{X \rightarrow A}\left(\Delta M^{\prime+}, M\right) \text { revise }_{X \rightarrow A}\left(\Delta M^{\prime-}, M\right)\right)
$$

$$
=\text { rrevise }_{X \rightarrow A}\left(\left(M^{\prime}, \Delta M^{\prime}\right),(M, \varnothing)\right) \quad \text { lemma } 27
$$

$$
\delta p u t_{\ell^{\prime}}(M, \Delta N)=\delta p u t_{\ell}(M, \Delta N)
$$

## Proof of Theorem 5

Lemma 90. If $M=\varnothing$ and $\Delta M$ is minimal for $M$, then $\dot{\pi}_{U}(M, \Delta M)=\left(\pi_{U}\left(\Delta M^{+}\right), \varnothing\right)$.

Proof.
$M=\varnothing$
suppose $M$ (1)
$\Delta M$ minimal for $M$
suppose $\Delta M$
$\Delta M^{-}=\varnothing$
lem. 23 (2)
$\dot{\pi}_{U}(M, \Delta M)$
$=\left(\pi_{U}\left(\Delta M^{+}\right)-\pi_{U}(M), \pi_{U}\left(\Delta M^{-}\right)-\pi_{U}(M \oplus \Delta M)\right)$
lem. 26 part 2
$=\left(\pi_{U}\left(\Delta M^{+}\right)-\varnothing, \varnothing-\pi_{U}(M \oplus \Delta M)\right)$
(1). 2); def. $\pi_{U}(\cdot)$
$=\left(\pi_{U}\left(\Delta M^{+}\right), \varnothing\right)$
def. -

Lemma 91. If $M, N=\varnothing$ and $\Delta M$ is minimal for $M$ and $\Delta N$ is minimal for $N$, then $(M, \Delta M) \dot{\cup}(N, \Delta N)=\left(\Delta M^{+} \cup \Delta N^{+}, \varnothing\right)$.

Proof. We need to prove $(M \oplus \Delta M) \cup(N \oplus \Delta N)=(M \cup N) \oplus\left(\Delta M^{+} \cup \Delta N^{+}, \varnothing\right)$.
$M=\varnothing \quad$ suppose $M$ (1)
$N=\varnothing$
suppose $N$ (2)
$\Delta M$ minimal for $M$
suppose $\Delta M$
$\Delta N$ minimal for $N$
suppose $\Delta N$
$\Delta M^{-}=\varnothing$
lem. 23 (3)
$\Delta N^{-}=\varnothing$
lem. 23 (4)
$(M \oplus \Delta M) \cup(N \oplus \Delta N)$
$=\left(\left(M \cup \Delta M^{+}\right)-\Delta M^{-}\right) \cup\left(\left(N \cup \Delta N^{+}\right)-\Delta N^{-}\right)$
$=\Delta M^{+} \cup \Delta N^{+}$
$=\varnothing \cup\left(\Delta M^{+} \cup \Delta N^{+}\right)$
$\varnothing$ unit for $\cup$
$=\left(\varnothing \cup\left(\Delta M^{+} \cup \Delta N^{+}\right)\right)-\varnothing \quad \varnothing$ unit for -
$=\varnothing \oplus\left(\Delta M^{+} \cup \Delta N^{+}, \varnothing\right)$
$=(M \cup N) \oplus\left(\Delta M^{+} \cup \Delta N^{+}, \varnothing\right)$
def. $\oplus$
(1. 2, 3. 4)
def. $\oplus$
$M \cup N=\varnothing$

Lemma 92. If $M=\varnothing$ and $\Delta M$ is minimal for $M$, then $\dot{\sigma}_{P}(M, \Delta M)=\left(\sigma_{P}\left(\Delta M^{+}\right), \varnothing\right)$.

Proof.
$M=\varnothing$
$\Delta M$ minimal for $M$
suppose $\Delta M$

$$
\begin{align*}
& \Delta M^{-}=\varnothing \\
& \dot{\sigma}_{P}(M, \Delta M) \\
& \quad=\left(\sigma_{P}\left(\Delta M^{+}\right), \sigma_{P}\left(\Delta M^{-}\right)\right) \\
& \quad=\left(\sigma_{P}\left(\Delta M^{+}\right), \sigma_{P}(\varnothing)\right)  \tag{1}\\
& \quad=\left(\sigma_{P}\left(\Delta M^{+}\right), \varnothing\right)
\end{align*}
$$

$$
\text { lem. } 23 \text { (1) }
$$

lem. 26 part 2
def. $\sigma_{P}(\cdot)$

Theorem 5. [Correctness of optimised join lens] Suppose $M: \operatorname{Rel}(U, P, F)$ and $N$ : $\operatorname{Rel}(V, Q, G)$ and $O=M \bowtie N$. Suppose also that $\Delta O$ is minimal with respect to $O$, and $O \oplus \Delta O: \operatorname{Rel}(U \cup V, P \bowtie Q, F \cup G)$. Then $\delta p u t_{\ell}((M, N), \Delta O)=\delta p u t_{\ell^{\prime}}((M, N), \Delta O)$.

Proof.
$O=M \bowtie N$
suppose $(O, M, N)$; def. get $_{\ell}$
$\left(M_{0}, \Delta M_{0}\right)=\operatorname{merge}_{F}^{\dagger}\left((M, \varnothing), \pi_{U}^{\dagger}(O, \Delta O)\right)$
$\left(N_{0}, \Delta N_{0}\right)=\operatorname{merge}_{G}^{\dagger}\left((N, \varnothing), \pi_{V}^{\dagger}(O, \Delta O)\right)$
$(L, \Delta L)=\left(\left(M_{0}, \Delta M_{0}\right) \bowtie^{\dagger}\left(N_{0}, \Delta N_{0}\right)\right)-^{\dagger}(O, \Delta O)$
$\left(L_{a}, \Delta L_{a}\right)=(L, \Delta L) \bowtie^{\dagger} \pi_{U \cap V}^{\dagger}(O, \Delta O)$
$\left(L_{l}, \Delta L_{l}\right)=(L, \Delta L)-^{\dagger}\left(L_{a}, \Delta L_{a}\right)$ suppose $\left(M_{0}, \Delta M_{0}, \Delta O\right)$
suppose $\left(N_{0}, \Delta N_{0}\right)$
suppose ( $L, \Delta L$ ) (3)
suppose ( $L_{a}, \Delta L_{a}$ ) (4)
suppose $\left(L_{l}, \Delta L_{l}\right)(5)$
suppose ( $\Delta M^{\prime}$ )
suppose $\left(\Delta N^{\prime}\right)$
def. $\delta p u t_{\ell}$
suppose $\left(P_{M}\right)$
suppose $\left(P_{N}\right)$
We derive the optimized expressions for $\Delta M_{0}$ and $\Delta N_{0}$ :

$$
\begin{array}{rlr}
M_{0}= & \operatorname{merge}_{F}\left(M, \pi_{U}(O)\right)=M & \text { def. . } ; \pi_{U}(O) \subseteq M \\
\begin{array}{rlr}
\Delta M_{0} & =\operatorname{dmerge}_{F}\left((M, \varnothing),\left(\pi_{U}(O), \dot{\pi}_{U}(O, \Delta O)\right)\right) & \text { def. . } \\
& =\operatorname{merge}_{F}\left(M, \dot{\pi}_{U}(O, \Delta O)^{+}\right) \ominus M & \text { lemma } 28 \\
& =\operatorname{merge}_{F}\left(M, \pi_{U}\left(\Delta O^{+}\right)-\pi_{U}(O)\right) \ominus M & \text { lemma } 26 \text { part } 2 \\
& =\operatorname{merge}_{F}\left(M, \pi_{U}\left(\Delta O^{+}\right)\right) \ominus M & \pi_{U}(O) \subseteq M \\
& =\operatorname{merge}_{F}\left(\sigma_{P_{M}}(M), \pi_{U}\left(\Delta O^{+}\right)\right) \ominus \sigma_{P_{M}}(M) & 7 ; \text { cor. } 3
\end{array}
\end{array}
$$

$$
\begin{array}{rlr}
N_{0}= & \text { merge }_{G}\left(N, \pi_{V}(O)\right)=N & \text { def. . }{ }^{\dagger} ; \pi_{V}(O) \subseteq N \\
\Delta N_{0} & =\operatorname{dmerge}_{G}\left((N, \varnothing),\left(\pi_{V}(O), \dot{\pi}_{V}(O, \Delta O)\right)\right) & \text { def. . } \dagger  \tag{9}\\
& =\operatorname{merge}_{G}\left(N, \dot{\pi}_{V}(O, \Delta O)^{+}\right) \ominus N & \text { lemma } 28 \\
& =\operatorname{merge}_{G}\left(N, \pi_{V}\left(\Delta O^{+}\right)-\pi_{V}(O)\right) \ominus N & \text { lem. } 26 \text { part } 2 \\
& =\operatorname{merge}_{G}\left(N, \pi_{V}\left(\Delta O^{+}\right)\right) \ominus N & \pi_{V}(O) \subseteq N \\
& =\operatorname{merge}_{G}\left(\sigma_{P_{N}}(N), \pi_{V}\left(\Delta O^{+}\right)\right) \ominus \sigma_{P_{N}}(N) & 8 ; \text { cor. } 3
\end{array}
$$

The unchanged set of records to be deleted $L$ is always empty:
$L=\left(M_{0} \bowtie N_{0}\right)-O=(M \bowtie N)-O=O-O=\varnothing$

We define $O \oplus \Delta O^{\prime}$ as the output view containing all functional dependency updates, but before deleting additional rows that may violate round-tripping guarantees.

$$
\begin{array}{rlrl}
\Delta O^{\prime} & =\left(M, \Delta M_{0}\right) \dot{凶}\left(N, \Delta N_{0}\right) & & \text { define }\left(\Delta O^{\prime}\right)  \tag{10}\\
\Delta O^{\prime+} & =\left(\left(M \oplus \Delta M_{0}\right) \bowtie \Delta N_{0}{ }^{+}\right) & \\
& \cup\left(\Delta M_{0}^{+} \bowtie\left(N \oplus \Delta N_{0}\right)\right) & & \text { lem. } 26 \text { part } 3 \\
\Delta O^{\prime-} & =\left(\Delta M_{0}^{-} \bowtie N\right) \cup\left(M \bowtie \Delta N_{0}^{-}\right) & & \text {lem. } 26 \text { part 3 }
\end{array}
$$

We then show that desired output view $O \oplus \Delta O$ is a subset of the previously defined view $O \oplus \Delta O^{\prime}$ :

$$
\begin{array}{lr}
M_{0} \oplus \Delta M_{0}=\text { merge }_{F}\left(M, \pi_{U}(O \oplus \Delta O)\right) & \text { (1); delta-correctness } \\
N_{0} \oplus \Delta N_{0}=\text { merge }_{G}\left(N, \pi_{V}(O \oplus \Delta O)\right) & \text { (2); delta-correctness } \\
\pi_{U}(O \oplus \Delta O) \subseteq M_{0} \oplus \Delta M_{0} & \text { def. } \text { merge }_{F}(\cdot, \cdot) \\
\pi_{V}(O \oplus \Delta O) \subseteq N_{0} \oplus \Delta N_{0} & \text { def. merge } G_{G}(\cdot, \cdot) \\
\pi_{U}(O \oplus \Delta O) \bowtie \pi_{V}(O \oplus \Delta O) & \bowtie \text { monotone } \\
\subseteq\left(M_{0} \oplus \Delta M_{0}\right) \bowtie\left(N_{0} \oplus \Delta N_{0}\right) & \bowtie \text { aft. } \pi_{U} \times \pi_{V} \text { incr.; trans. } \\
O \oplus \Delta O \subseteq\left(M_{0} \oplus \Delta M_{0}\right) \bowtie\left(N_{0} \oplus \Delta N_{0}\right) & 10] ; \text { delta-correctness }
\end{array}
$$

We now derive the expression for $\Delta L$ :
$M_{0} \bowtie N_{0}=M \bowtie N=O$
$\Delta L$

$$
=\left(\left(M_{0}, \Delta M_{0}\right) \bowtie^{\dagger}\left(N_{0}, \Delta N_{0}\right)\right) \dot{-}(O, \Delta O)
$$

$$
\begin{align*}
& =\left(O, \Delta O^{\prime}\right) \dot{-}(O, \Delta O)  \tag{12,10}\\
& =\Delta O^{\prime} \ominus \Delta O \\
& =\left(\left(\left(M \oplus \Delta M_{0}\right) \bowtie \Delta N_{0}^{+}\right) \cup\left(\Delta M_{0}^{+} \bowtie\left(N \oplus \Delta N_{0}\right)\right),\right. \\
& \left.\quad\left(\Delta M_{0}^{-} \bowtie N\right) \cup\left(M \bowtie \Delta N_{0}^{-}\right)\right) \ominus \Delta O
\end{align*}
$$

We now derive some straightforward properties required later in the proof, including that $L$ projected onto $U$ is a subset of $M_{0}$ and that $L \oplus \Delta L$ is a subset of $M_{0} \oplus M_{0}$.

$$
\begin{array}{lr}
\pi_{U}(L)=\varnothing \subseteq M_{0} & L=\varnothing ; \varnothing \text { least }(13) \\
\pi_{U}(L) \oplus \dot{\pi}_{U}(L, \Delta L) & \\
\quad=\pi_{U}(L \oplus \Delta L) & \text { delta-correctness } \\
\quad=\pi_{U}\left(\left(\left(M_{0} \oplus \Delta M_{0}\right) \bowtie\left(N_{0} \oplus \Delta N_{0}\right)\right)-(O \oplus \Delta O)\right) & \text { (33); delta-correctness } \\
\subseteq \pi_{U}\left(\left(M_{0} \oplus \Delta M_{0}\right) \bowtie\left(N_{0} \oplus \Delta N_{0}\right)\right) & \pi_{U}(\cdot) \text { monotone } \\
\subseteq M_{0} \oplus \Delta M_{0} & \pi_{U} \text { after } \bowtie \text { decr. }(14)
\end{array}
$$

We also show that $L_{a}$ and $L_{l}$ are empty, as well as that $L_{a} \oplus \Delta L_{a}$ and $L_{l} \oplus \Delta L_{l}$ are both subsets of $L \oplus \Delta L$ :

$$
\begin{align*}
& L_{a}=L \bowtie \pi_{U \cap V}(O)=\varnothing  \tag{15}\\
& L=\varnothing \\
& L=\varnothing \\
& L_{l}=L-L_{a}=\varnothing  \tag{16}\\
& L_{a} \oplus \Delta L_{a} \\
& =\left(L \bowtie \pi_{U}(O)\right) \oplus\left((L, \Delta L) \dot{\bowtie} \pi_{U \cap V}^{\dagger}(O, \Delta O)\right) \\
& \exp . L_{a}, \Delta L_{a} \\
& =(L \oplus \Delta L) \bowtie\left(\pi_{U \cap V}(O) \oplus \dot{\pi}_{U \cap V}(O \oplus \Delta O)\right) \\
& \text { delta-correctness } \\
& =\pi_{U \cup V}\left((L \oplus \Delta L) \bowtie\left(\pi_{U \cap V}(O) \oplus \dot{\pi}_{U \cap V}(O \oplus \Delta O)\right)\right) \\
& \pi \text { identity } \\
& \subseteq L \oplus \Delta L \\
& \pi_{U \cup V} \text { after } \bowtie \text { decr. }  \tag{17}\\
& \sigma_{P_{d}}\left(L_{a}\right) \oplus \dot{\sigma}_{P_{d}}\left(L_{a}, \Delta L_{a}\right) \\
& =\sigma_{P_{d}}\left(L_{a} \oplus \Delta L_{a}\right) \quad \text { delta-correctness } \\
& \subseteq L_{a} \oplus \Delta L_{a} \quad \sigma_{P_{d}}(\cdot) \text { decreasing } \\
& \subseteq L \oplus \Delta L \\
& \text { (17); } \subseteq \text { transitive (18) } \\
& L_{l} \oplus \Delta L_{l} \\
& =\left(L-L_{a}\right) \oplus\left((L, \Delta L) \dot{-}\left(L_{a}, \Delta L_{a}\right)\right) \\
& =(L \oplus \Delta L)-\left(L_{a} \oplus \Delta L_{a}\right) \quad \text { delta-correctness } \\
& \subseteq(L \oplus \Delta L)  \tag{19}\\
& \text { - decreasing }
\end{align*}
$$

The following few properties are required later in the proof:

$$
\begin{aligned}
& \left(L_{l} \cup \sigma_{P_{d}}\left(L_{a}\right)\right) \oplus\left(\left(L_{l}, \Delta L_{l}\right) \dot{\cup}\left(\sigma_{P_{d}}^{\dagger}\left(L_{a}, \Delta L_{a}\right)\right)\right) \\
& \quad=\left(L_{l} \oplus \Delta L_{l}\right) \cup\left(\sigma_{P_{d}}\left(L_{a}\right)\right. \\
& \left.\quad \oplus \sigma_{P_{d}}\left(L_{a}, \Delta L_{a}\right)\right) \\
& \quad \subseteq L \oplus \Delta L
\end{aligned}
$$

$$
\left.\oplus \sigma_{P_{d}}\left(L_{a}, \Delta L_{a}\right)\right) \quad \text { delta-correctness }
$$

(18, 19); $\cup$ least up. bound (20)

$$
\begin{array}{lr}
\pi_{U}\left(L_{l} \cup \sigma_{P_{d}}\left(L_{a}\right)\right) \oplus \dot{\pi}_{U}\left(\left(L_{l}, \Delta L_{l}\right) \cup^{\dagger} \sigma_{P_{d}}^{\dagger}\left(L_{a}, \Delta L_{a}\right)\right) & \text { delta-correctness } \\
\quad=\pi_{U}\left(\left(L_{l} \cup \sigma_{P_{d}}\left(L_{a}\right)\right) \oplus\left(\left(L_{l}, \Delta L_{l}\right) \dot{\cup}\left(\sigma_{P_{d}}^{\dagger}\left(L_{a}, \Delta L_{a}\right)\right)\right)\right) & \text { 20); } \pi_{U}(\cdot) \text { monotone }
\end{array}
$$

$$
\subseteq M_{0} \oplus \Delta M_{0}
$$

(14); $\subseteq$ transitive (21)

$$
\sigma_{P_{d}}\left(L_{a}\right)=\varnothing
$$

(15); $\varnothing$ least (22)

$$
\pi_{U}\left(L \cup \sigma_{P_{d}}\left(L_{a}\right)\right)=\varnothing \subseteq M_{0}
$$

(16. 22); $\varnothing$ least (23)

We can now derive expressions for $\Delta L_{a}{ }^{+}, \Delta L_{l}{ }^{+}$and $\Delta M^{\prime}$ :
$\Delta L_{a}^{+}$

$$
\begin{aligned}
& =L_{a} \oplus \Delta L_{a}^{+} \\
& =\left(L \bowtie \pi_{U \cap V}(O)\right) \oplus(L, \Delta L) \dot{\bowtie} \pi_{U \cap V}^{\dagger}(O, \Delta O) \\
& =(L \oplus \Delta L) \bowtie \pi_{U \cap V}(O \oplus \Delta O)
\end{aligned}
$$

(15); lem. 23
def. $L \oplus \Delta L_{a}^{+}$ delta-correctness
$\Delta L_{l}$
$=(L, \Delta L) \dot{-}\left(L_{a}, \Delta L_{a}\right)$
$=\Delta L \ominus \Delta L_{a}$
$=\Delta L \oplus\left(\Delta L_{a}^{-}, \Delta L_{a}^{+}\right)$
def. $\ominus$
$\Delta L_{l}^{+}$

$$
\begin{array}{lr}
=\left(\Delta L^{+}-\Delta L_{a}^{+}\right) \cup\left(\Delta L_{a}^{-}-\Delta L^{-}\right) & (24) ; \text { def. } \oplus \\
=\Delta L^{+}-\Delta L_{a}^{+} & \Delta L_{a}^{-}=\varnothing
\end{array}
$$

$$
\begin{aligned}
& \Delta M^{\prime} \\
& \quad=\left(M_{0}, \Delta M_{0}\right) \dot{-}\left(\pi_{U}^{\dagger}\left(\left(L_{l}, \Delta L_{l}\right) \cup^{\dagger} \sigma_{P_{d}}^{\dagger}\left(L_{a}, \Delta L_{a}\right)\right)\right)
\end{aligned}
$$

After showing some helper properties, we derive an expression for $\Delta N^{\prime}$ :

$$
=\Delta M_{0} \ominus\left(\dot{\pi}_{U}\left(\left(L_{l}, \Delta L_{l}\right) \cup^{\dagger} \sigma_{P_{d}}^{\dagger}\left(L_{l}, \Delta L_{l}\right)\right)\right)
$$

$$
\begin{equation*}
=\Delta M_{0} \ominus\left(\pi_{U}\left(\Delta L_{l}^{+} \cup \sigma_{P_{d}}\left(\Delta L_{a}^{+}\right)\right)\right) \tag{22,23}
\end{equation*}
$$

$$
\sigma_{Q_{d}}\left(L_{a}\right)=\varnothing
$$

(15); def. $\sigma_{Q_{d}}(\cdot)(26)$

$$
\pi_{V}\left(\sigma_{Q_{d}}\left(L_{a}\right)\right)=\varnothing \subseteq N_{0}
$$

(26); $\varnothing$ least (27)

$$
\pi_{V}(L)=\varnothing \subseteq N_{0}
$$

$L=\varnothing, \varnothing$ least (28)

$$
\pi_{V}(L) \oplus \dot{\pi}_{V}(L, \Delta L)
$$

$$
=\pi_{V}(L \oplus \Delta L)
$$

delta-correctness

$$
=\pi_{V}\left(\left(\left(M_{0} \oplus \Delta M_{0}\right) \bowtie\left(N^{\prime} \oplus \Delta N^{\prime}\right)\right)-(O \oplus \Delta O)\right)
$$

(3); delta-correctness

$$
\subseteq \pi_{V}\left(\left(M_{0} \oplus \Delta M_{0}\right) \bowtie\left(N^{\prime} \oplus \Delta N^{\prime}\right)\right)
$$

$$
\begin{equation*}
\subseteq N_{0} \oplus \Delta N_{0} \tag{29}
\end{equation*}
$$

$$
\begin{array}{rlr}
\sigma_{Q_{d}}\left(L_{a}\right) \oplus \dot{\sigma}_{Q_{d}}\left(L_{a}, \Delta L_{a}\right) & \\
& =\sigma_{Q_{d}}\left(L_{a} \oplus \Delta L_{a}\right) & \text { delta-correctness } \\
& \subseteq L_{a} \oplus \Delta L_{a} & \\
& \subseteq L \oplus \Delta L & \\
\sigma_{Q_{d}}(\cdot) \text { decreasing }
\end{array}
$$

$$
\pi_{V}\left(\sigma_{Q_{d}}\left(L_{a}\right)\right) \oplus \dot{\pi}_{V}\left(\sigma_{Q_{d}}^{\dagger}\left(L_{a}\right)\right)
$$

$$
=\pi_{V}\left(\sigma_{Q_{d}}\left(L_{a}\right) \oplus \dot{\sigma}_{Q_{d}}\left(L_{a}, \Delta L_{a}\right)\right)
$$

$$
\subseteq \pi_{V}(L \oplus \Delta L)
$$

$$
\subseteq N_{0} \oplus \Delta N_{0}
$$

$\Delta N^{\prime}$
$=\left(N_{0}, \Delta N_{0}\right) \doteq\left(\pi_{V}^{\dagger}\left(\sigma_{Q_{d}}^{\dagger}\left(L_{a}, \Delta L_{a}\right)\right)\right)$
$=\Delta N_{0} \ominus\left(\dot{\pi}_{V}\left(\sigma_{Q_{d}}^{\dagger}\left(L_{a}, \Delta L_{a}\right)\right)\right) \quad$ 27, 31); lem. 26 part 5
$=\Delta N_{0} \ominus\left(\pi_{V}\left(\sigma_{Q_{d}}\left(\Delta L_{a}^{+}\right)\right), \varnothing\right)$
$\delta p u t_{\ell^{\prime}}((M, N), \Delta O)=\left(\Delta M^{\prime}, \Delta N^{\prime}\right)$
$\delta p u t_{\ell^{\prime}}((M, N), \Delta O)=\delta p u t_{\ell}((M, N), \Delta O)$
(26) 27); lem. 90.92
def. $\delta p u t_{\ell^{\prime}}$
(32, 6); transitivity

## Proof of Theorem 6

Theorem 6. [Correctness of rename lens] Suppose $M: \operatorname{Rel}(U, P, F)$ and $N=\rho_{A / B}(M)$, and that $\Delta N$ is minimal with respect to $N$ and satisfies $N \oplus \Delta N: \operatorname{Rel}(U \cup V, P \bowtie$ $Q, F \cup G)$. Then $\delta p u t_{\ell}(M, \Delta N)=\delta p u t_{\ell^{\prime}}(M, \Delta N)$.

Proof.

$$
\begin{aligned}
& N=\operatorname{get}_{\ell}(M) \\
& \left(M^{\prime}, \Delta M^{\prime}\right)=\rho_{B / A}^{\dagger}(N, \Delta N) \\
& =\left(\rho_{B / A}(N), \dot{\rho}_{B / A}(N, \Delta N)\right) \\
& \delta p u t_{\ell}=\Delta M^{\prime} \\
& \Delta M^{\prime}=\dot{\rho}_{B / A}(N, \Delta N) \\
& =\left(\rho_{B / A}\left(\Delta N^{+}\right), \rho_{B / A}\left(\Delta N^{-}\right)\right) \\
& \delta p u t_{\ell^{\prime}}=\Delta M^{\prime} \\
& =\left(\rho_{B / A}\left(\Delta N^{+}\right), \rho_{B / A}\left(\Delta N^{-}\right)\right) \\
& \delta p u t_{\ell}=\delta p u t_{\ell^{\prime}} \\
& \text { suppose ( } N, M \text { ) } \\
& \text { suppose ( } N_{\text {new }}, \Delta N_{\text {new }}, \Delta N \text { ) } \\
& \text { def. }{ }^{\dagger} \text { (1) } \\
& \text { def. } \delta_{\text {put }}^{\ell} \text { (2) } \\
& \text { (1) } \\
& \text { lem. } 26 \text { part } 4 \text { (3) } \\
& \text { def. } \delta p u t_{\ell^{\prime}} \\
& \text { (3) (4) } \\
& \text { (4. 2); transitivity }
\end{aligned}
$$

## Appendix C

## Proofs for Chapter 4

## C. 1 Normalisation Proofs 4.3

Proposition 9 (Normal forms). If $x: R \vdash e: \tau$ and $e \rightsquigarrow{ }^{*} f \nsim$, then $f$ is in normal form.

Proof. As the rewrite rules can be applied anywhere in a term, it follows that if we cannot apply a normalisation rule to a term, then we cannot apply a normalisation rule to any of its subterms.

Terms typeable by rules T-VAR, T-Const, and T-ABS are already in normal form. Rules T-Record and T-Op follow directly from the induction hypothesis. The remainder of the cases follow.

Case T-App

Assumption:

$$
\frac{x: R \vdash e: A \rightarrow B \quad x: R \vdash f: A}{x: R \vdash e f: B}
$$

By the induction hypothesis, we have that $e$ and $f$ are in normal form.
Given that $e$ is in normal form and has function type, there are the following possibilities:

- $e=x$, which is not possible since the only variable in the typing environment is $x$, and $R$ is not a function type
- $e=c$, which is not possible since constants only have base types, not function types
- $e=\lambda x . e^{\prime}$. In this case, we could apply the first normalisation rule, which would be a contradiction.
- $e=x . \ell$, which is not possible since $R$ only contains fields with base types, not function types
- if $v_{1}$ then $v_{2}$ else $v_{3}$. In this case, we could apply the fifth normalisation rule, which would be a contradiction.
- $\odot\{\vec{v}\}$, which is not possible since the result of an operator must have base type. Thus, a term $x: R \vdash e f: \tau$ cannot be in normal form.


## Case T-Project

assumption:

$$
\frac{\Gamma \vdash e:\left(\left(\ell_{i}: A_{i}\right)_{i \in I}\right) \quad j \in I}{\Gamma \vdash e . \ell_{j}: A_{j}}
$$

By the IH , we have that $e$ is in normal form. We now perform case analysis on $e$, giving us the following possibilities for terms in normal form which can have record type:

- $e=x$ : We have that $x . \ell$ which is in normal form.
- $e=(\overrightarrow{\ell=v})$ : Impossible, since it would be possible to reduce by the second normalisation rule
- $e=$ if $v_{1}$ then $v_{2}$ else $v_{3}$ : Impossible, since it would be possible to reduce by the sixth reduction rule


## Case T-If

Assumption:

$$
\frac{\Gamma \vdash e: \text { bool } \quad \Gamma \vdash f_{1}: A \quad \Gamma \vdash f_{2}: A}{\Gamma \vdash \text { if } e \text { then } f_{1} \text { else } f_{2}: A}
$$

Immediate by the induction hypothesis on all three subterms; normalisation rules 3 and 4 serve only as an optimisation.

## C. 2 Hybrid Predicates (Section 4.4.1)

Lemma 51 (Consistent Typing). If $R \vdash S: A$ and $P:: S$ then $R \vdash P: A$.

Proof.
$R \vdash S: A \quad$ assumption
$P:: S$
assumption (1)

Perform induction on $R \vdash S: A$.
case: T-S-ConsT

$$
\frac{c: \tau_{c}}{R \vdash c: \tau_{c}}
$$

assumption
by inversion on $P:: c$
$c:: c$
C-Const (1)
$R \vdash c: \tau_{c}$
T-Const
case: T-S-Label

$$
\frac{(\ell: A) \in R}{R \vdash \ell: A}
$$

assumption
by inversion on $P:: \ell$
$\ell:: \ell$
C-Label (1)
$R \vdash \ell: \tau_{c}$
T-Label
case: T-S-IF

$$
\begin{gathered}
R \vdash S_{1}: \text { bool } \\
\frac{R \vdash T_{1}: A \quad R \vdash T_{2}: A}{R \vdash \text { if } S_{1} \text { then } T_{1} \text { else } T_{2}: A}
\end{gathered}
$$

by inversion on $P$ :: if $S_{1}$ then $T_{1}$ else $T_{2}$
$P_{1}:: S \quad Q_{1}:: T_{1} \quad Q_{2}:: T_{2}$
C-IF (1)
if $P_{1}$ then $Q_{1}$ else $Q_{2}::$ if $S$ then $T_{1}$ else $T_{2}::$
$R \vdash P_{1}$ : bool IH
$R \vdash Q_{1}: A$
IH
$R \vdash Q_{2}: A$
IH
$R \vdash$ if $P_{1}$ then $P_{1}$ else $Q_{2}: A$
case: T-S-Op

$$
\frac{\odot: A_{1} \times \ldots \times A_{n} \rightarrow A \quad\left(R \vdash S_{i}: A_{i}\right)_{i \in 1 . . n}}{R \vdash \odot \vec{S}: A}
$$

assumption
by inversion on $P:: \odot \vec{S}$
$\frac{\odot: A_{1} \times \ldots \times A_{n} \rightarrow A \quad\left(P_{i}:: S_{i}\right)_{i \in 1 . . n}}{\odot\{\vec{P}\}:: \odot \vec{S}}$

$\left(R \vdash P_{i}: A_{i}\right)_{i \in 1 . . n}$
IH
$R \vdash \odot \vec{P}: A$
case: T-S-ERased

$$
\frac{R_{1} \subseteq R}{R \vdash\left(R_{1} \vdash ?: A\right): A}
$$

assumption
by inversion on $P::\left(R_{1} \vdash ?: A\right)$
$\frac{R_{1} \vdash P: A}{P::\left(R_{1} \vdash ?: A\right)}$
$R \vdash P: A$
C-Erased (1)

Lemma 35

Lemma 52 (Static predicate consistent). For any closed term e (or term ê) such that $\cdot \vdash e:$ pred $S$ and $R \vdash S: A$, if $e \Downarrow P$ then $P:: S$.

Proof.
$\cdot \vdash e:$ pred $S$ assumption
$R \vdash S: A$
assumption (1)
$e \Downarrow P$
assumption (2)

Perform induction on $e: \operatorname{pred} S$ :
case: T-Q-Const
$c:$ pred $c$
by inversion on $c \Downarrow P$
$c \Downarrow c$
$c:: c$
E-Q-Const (2)
C-Const
case: T-Q-Label
$\overline{\ell: \text { pred } \ell}$
by inversion on $\ell \Downarrow P$
$\ell \Downarrow \ell$
$\ell:: \ell$
E-Q-Label (2)
C-Label
case: T-Q-IF-ElsE
$\qquad$ assumption
if $\hat{e}$ then $\hat{f}_{1}$ else $\hat{f}_{2}:$ pred if $S_{1}$ then $T_{1}$ else $T_{2}$
by inversion on $R \vdash$ if $S_{1}$ then $T_{1}$ else $T_{2}$

$$
R \vdash S_{1}: \text { bool }
$$

$\frac{R \vdash T_{1}: A \quad R \vdash T_{2}: A}{R \vdash \text { if } S_{1} \text { then } T_{1} \text { else } T_{2}: A}$
T-S-IF (1)
by inversion on if $\hat{e}$ then $\hat{f}_{1}$ else $\hat{f}_{2} \Downarrow P$
$\hat{e} \Downarrow P_{1}$
$\frac{\hat{f}_{1} \Downarrow Q_{1} \quad \hat{f}_{2} \Downarrow Q_{2}}{\text { if } \hat{e}_{1} \text { then } \hat{f}_{1} \text { else } \hat{f}_{2} \Downarrow \text { if } P_{1} \text { then } Q_{1} \text { else } Q_{2}}$
E-If-Else (2)
$P_{1}:: S_{1}$
$Q_{1}:: T_{1}$
$Q_{2}:: T_{2}$
if $P_{1}$ then $Q_{1}$ else $Q_{2}::$ if $S_{1}$ then $T_{1}$ else $T_{2}$
case: T-Q-OP

$$
\frac{\odot: A_{1} \times \ldots \times A_{n} \rightarrow A \quad\left(\hat{e}_{i}: \text { pred } S_{i}\right)_{i \in 1 . . n}}{\odot \overrightarrow{\hat{e}}: \text { pred } \odot \vec{S}}
$$

assumption
by inversion on $R \vdash \odot \vec{S}: A$
$\frac{\odot: A_{1} \times \ldots \times A_{n} \rightarrow A \quad\left(R \vdash S_{i}: A_{i}\right)_{i \in 1 . . n}}{R \vdash \odot \vec{S}: A}$
T-S-Op (1)
by inversion on $\odot \vec{e} \Downarrow P$
$\frac{\left(\hat{e}_{i} \Downarrow Q_{i}\right)_{i \in 1 . . n}}{\odot \vec{e} \Downarrow \odot\{\vec{Q}\}}$
E-Q-Op (2)
$\left(Q_{i}:: S_{i}\right)_{i \in 1 . . n}$
IH
$\odot\{\vec{Q}\}:: \odot \vec{S}$
C-Op
case: T-H-Erase
$\frac{R_{1} \vdash T: A \quad f: \text { pred } T}{\text { erase } f: \operatorname{pred}\left(R_{1} \vdash \sqrt{?}: A\right)}$
assumption
by inversion on erase $f \Downarrow P$
$\frac{f \Downarrow P}{\text { erase } f \Downarrow P}$
E-H-Erase (2)
$P:: T$
IH
$\begin{array}{lc}R_{1} \vdash P: A & \text { Lemma 51 } \\ P::\left(R_{1} \vdash ?: A\right) & \text { C-ERASED }\end{array}$

## Appendix D

## Proofs for Chapter 5

## D. 1 Language is well-behaved

We first define substitution in Figure D.1.

$$
\begin{aligned}
c[v / x] & =c \\
x[v / x] & =v \\
y[v / x] & =w \\
\lambda x \cdot e[v / x] & =\lambda x \cdot e \\
\lambda y \cdot e[v / x] & =\lambda y \cdot e[v / x] \\
(e f)[v / x] & =(e[v / x])(f[v / x]) \\
\{\vec{e}\}[v / x] & =\{\overrightarrow{e[v / x]}\} \\
(\overrightarrow{\ell=e})[v / x] & =(\overrightarrow{\ell=e[v / x]})
\end{aligned}
$$

(lens $S$ of $R$ with $F)[v / x]=$ lens $S$ of $R$ with $F$
$\left(\operatorname{select}_{Q}\right.$ from $\left.e\right)[v / x]=\operatorname{select}_{Q}$ from $e[v / x]$
(drop $\ell$ determined by $(U, v)$ from $e)[v / x]=$ drop $\ell$ determined by $(U, v)$ from $e[v / x]$
$\left(\right.$ rename $_{\ell / \ell^{\prime}}$ from $\left.e\right)[v / x]=\operatorname{rename}_{\ell / \ell^{\prime}}$ from $e[v / x]$
(get $e)[v / x]=$ get $e[v / x]$
(put $e$ with $f$ ) $[v / x]=$ put $e[v / x]$ with $f[v / x]$

Figure D.1: Substitution on terms

Lemma 93 (weakening). If $\Gamma \vdash e: \tau$ and $\Gamma$ is a subset of $\Gamma^{\prime}$ then $\Gamma^{\prime} \vdash e: \tau$.

Proof.
$\Gamma \vdash e: \tau$
assumption
$\Gamma \subseteq \Gamma^{\prime}$
assumption
perform induction on $\Gamma \vdash e: \tau$
case T-Const:
$c$ is of type $A$

$$
\Gamma \vdash c: A
$$

$\Gamma^{\prime} \vdash c: A$
assumption

T-Const
case T-VAR:

$$
\frac{\left(x=\tau_{1}\right) \in \Gamma}{\Gamma \vdash x: \tau_{1}}
$$

assumption

$$
\left(x=\tau_{1}\right) \in \Gamma^{\prime}
$$

$\Gamma \subseteq \Gamma^{\prime}$

$$
\Gamma \vdash x: \tau_{1}
$$

T-VAR
case T-ABS:

$$
\begin{aligned}
& \frac{\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash f: \tau_{1}}{\Gamma \vdash \lambda x . f: \tau_{1} \rightarrow \tau_{2}} \\
& \Gamma \longleftarrow\left(x=\tau_{1}\right) \subseteq \Gamma^{\prime} \leftarrow\left(x=\tau_{1}\right) \\
& \Gamma^{\prime} \longleftarrow\left(x=\tau_{1}\right) \vdash f: \tau_{1}\left(x=\tau_{1}\right) \\
& \Gamma^{\prime} \vdash \lambda x . f: \tau_{1} \rightarrow \tau_{2}
\end{aligned}
$$

case T-App:

$$
\begin{aligned}
& \frac{\Gamma \vdash e_{1}: \tau_{1} \rightarrow \tau_{2} \quad \Gamma \vdash f: \tau_{1}}{\Gamma \vdash e_{1} f: \tau_{2}} \\
& \Gamma^{\prime} \vdash e_{1}: \tau_{1} \rightarrow \tau_{2} \\
& \Gamma^{\prime} \vdash f: \tau_{1} \\
& \Gamma^{\prime} \vdash e_{1} f: \tau_{2}
\end{aligned}
$$

assumption induction induction T-App
case T-RECORD:

$$
\begin{aligned}
& \frac{\left(\Gamma \vdash e_{i}: \tau_{i}\right)_{i \in 1 \ldots n}}{\Gamma \vdash(\overrightarrow{\ell=e}):(\overrightarrow{\ell: \tau})} \\
& \left(\Gamma^{\prime} \vdash e_{i}: \tau_{i}\right)_{i \in 1 \ldots n} \\
& \Gamma^{\prime} \vdash(\overrightarrow{\ell=e}):(\overrightarrow{\ell: \tau})
\end{aligned}
$$

case T-Const-Set:

$$
\begin{aligned}
& \frac{\left(\Gamma \vdash e_{i}: \tau\right)_{i \in 1 \ldots n}}{\Gamma \vdash\{\vec{e}\}:\{\tau\}} \\
& \left(\Gamma^{\prime} \vdash e_{i}: \tau\right)_{i \in 1 \ldots n} \\
& \Gamma^{\prime} \vdash\{\vec{e}\}:\{\tau\}
\end{aligned}
$$ assumption

$$
\left(\Gamma^{\prime} \vdash e_{i}: \tau\right)_{i \in 1 \ldots n} \quad \text { induction }
$$ T-Const-SET

case T-Lens:
$\frac{(S=\operatorname{Rel}(R, \text { true }, F)) \in \Phi \quad \bigcup \operatorname{names}(F) \subseteq \operatorname{dom}(R)}{\Gamma \vdash \text { lens } S \text { of } R \text { with } F: \text { lens of }(\{S\}, R, \text { true }, F)} \quad$ assumption
case T-Select:

| $\Gamma \vdash f:$ lens of $(\Sigma, R, P, F) \quad R \vdash Q:$ bool |  |
| :--- | ---: |
| $F$ is in tree form $\quad P$ ignores outputs $(F)$ |  |
| $\Gamma \vdash \operatorname{select}_{Q}$ from $f:$ lens of $(\Sigma, R, P \wedge Q, F)$ | assumption |
| $\Gamma^{\prime} \vdash f:$ lens of $(\Sigma, R, P, F)$ | induction |
| $\Gamma^{\prime} \vdash \operatorname{select}_{Q}$ from $f:$ lens of $(\Sigma, R, P \wedge Q, F)$ | T-Select |

case T-Drop:

$$
\begin{gathered}
F \equiv G \cup\{U \rightarrow \ell\} \quad \Gamma \vdash f: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P, F\right) \\
U \subseteq \operatorname{dom}(R) \quad \cdot \vdash v: A
\end{gathered}
$$

$$
\mathbf{L J D}[R,(\ell: A)](P) \quad \mathbf{D V}[R,(\ell: A)](P,(\ell=v))
$$

$\Gamma \vdash$ drop $\ell$ determined by $(U, v)$ from $f:$ lens of $\left(\Sigma, R, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right)$
assumption
$\Gamma^{\prime} \vdash f:$ lens of $\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P, F\right)$ induction
$\Gamma^{\prime} \vdash$ drop $\ell$ determined by $(U, v)$ from $f$ :lens of $\left(\Sigma, R, \llbracket \rrbracket_{\rrbracket_{1}, R_{2}}, G\right) \quad$ T-Drop case T-Join-Templ:
$\Gamma \vdash e_{1}:$ lens of $(\Sigma, R, P, F)$
$\Gamma \nvdash \vdash e_{2}:$ lens of $\left(\Delta, R^{\prime}, Q, G\right) \quad \Gamma \vdash f: R \oplus R^{\prime} \rightarrow(\nwarrow|\uparrow| \nearrow)$
$G \vDash \operatorname{dom}(R) \cap \operatorname{dom}\left(R^{\prime}\right) \rightarrow \operatorname{dom}\left(R^{\prime}\right) \quad F$ is in tree form $\quad G$ is in tree form
$P$ ignores outputs $(F) \quad Q$ ignores outputs $(G) \quad \Sigma \cap \Delta=\varnothing$
$\Gamma \vdash$ join $_{f} e_{1}$ with $e_{2}:$ lens of $\left(\Sigma \cup \Delta, R \oplus R^{\prime}, P \wedge Q, F \cup G\right)$
assumption
$\Gamma^{\prime} \vdash e_{1}:$ lens of $(\Sigma, R, P, F)$ induction

```
\(\Gamma^{\prime} \vdash e_{2}\) :lens of \(\left(\Delta, R^{\prime}, Q, G\right)\)
induction
\(\Gamma^{\prime} \vdash f: R \oplus R^{\prime} \rightarrow(\nwarrow|\uparrow| \nearrow) \quad\) induction
\(\Gamma^{\prime} \vdash \mathbf{j o i n}_{f} e_{1}\) with \(e_{2}:\) lens of \(\left(\Sigma \cup \Delta, R \oplus R^{\prime}, P \wedge Q, F \cup G\right)\)
case T-Rename:
\(\frac{\Gamma \vdash f: \text { lens of }(\Sigma, R \oplus(\ell: A), P, F)}{\Gamma \vdash \text { rename }_{\ell / \ell^{\prime}} \text { from } f: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P\left[\ell / \ell^{\prime}\right], F\left[\ell / \ell^{\prime}\right]\right)}\)
assumption
\(\begin{array}{lr}\Gamma^{\prime} \vdash f: \text { lens of }(\Sigma, R \oplus(\ell: A), P, F) & \text { induction } \\ \Gamma^{\prime} \vdash \text { rename }_{\ell / \ell^{\prime}} \text { from } f: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P\left[\ell / \ell^{\prime}\right], F\left[\ell / \ell^{\prime}\right]\right) & \text { T-RENAME }\end{array}\) case T-GET:
```

$\Gamma \vdash f:$ lens of $(\Sigma, R, P, F)$
$\Gamma \vdash$ get $f:\{R\}$
$\Gamma^{\prime} \vdash f:$ lens of $(\Sigma, R, P, F) \quad$ induction
$\Gamma^{\prime} \vdash$ get $f:\{R\}$
assumption
case T-Put:

| $\Gamma \vdash e_{1}:$ lens of $(\Sigma, R, P, F) \quad \Gamma \vdash f:\{R\}$ |  |
| :---: | :---: |
| $\Gamma \vdash$ put $e_{1}$ with $f$ : bool |  |
| $\Gamma^{\prime} \vdash e_{1}$ : lens of $(\Sigma, R, P, F)$ | induction |
| $\Gamma^{\prime} \vdash f:\{R\}$ | induction |
| $\Gamma^{\prime} \vdash$ put $e_{1}$ with $f$ : bool | T-Put |

Lemma 67 (Substitution). If $\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash e: \tau$ and $\cdot \vdash v: \tau_{1}$ then $\Gamma \vdash e[v / x]: \tau$.

Proof.
$\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash e: \tau \quad$ assumption
$\cdot \vdash v: \tau_{1} \quad$ assumption (1)
Perform induction on $\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash e: \tau$
case T-Const:

$$
\frac{c \text { is of type } A}{\Gamma \leftarrow\left(x=\tau_{1}\right) \vdash c: A}
$$

assumption
$c[v / x]=c$
def. subst
T-Const
case T-VAR:

$$
\frac{\left(x=\tau_{1}\right) \in \Gamma \nleftarrow\left(x=\tau_{1}\right)}{\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash y: \tau_{1}}
$$

assumption (2)
if $x=y$ :

$$
\begin{aligned}
& x[v / x]=v \\
& \Gamma \vdash v: \tau_{1}
\end{aligned}
$$

else $x \neq y$ :

$$
\begin{array}{lr}
y[v / x]=y & \text { def. subst } \\
\left(y=\tau_{1}\right) \in \Gamma & (2)  \tag{2}\\
\Gamma \vdash y: \tau_{1} & \text { T-VAR }
\end{array}
$$

def. subst
(1); weakening
case T-ABS:

$$
\frac{\left(\Gamma \leftarrow\left(x=\tau_{1}\right)\right) \leftarrow\left(y=\tau_{3}\right) \vdash f: \tau_{2}}{\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash \lambda y \cdot f: \tau_{3} \rightarrow \tau_{2}}
$$

assumption (3)
if $x=y$ :
$(\lambda x . f)[v / x]=\lambda x . f$

$$
\left(\Gamma \leftarrow\left(x=\tau_{1}\right)\right) \leftarrow\left(x=\tau_{3}\right)=\Gamma \leftarrow\left(x=\tau_{3}\right)
$$

$\Gamma \longleftarrow\left(x=\tau_{3}\right) \vdash \lambda x . f: \tau_{2}$
$\Gamma \vdash \lambda y . f: \tau_{3} \rightarrow \tau_{2}$
def. subst
overwrites (4)
else $x \neq y$ :
( $\lambda y . f)[v / x]=\lambda y .(f[v / x])$
$\left(\Gamma \leftarrow\left(x=\tau_{1}\right)\right) \leftarrow\left(y=\tau_{3}\right)$ $=\left(\Gamma \longleftarrow\left(y=\tau_{3}\right)\right) \longleftarrow\left(x=\tau_{1}\right)$
$\Gamma \leftrightarrow\left(y=\tau_{3}\right) \vdash f[v / x]: \tau_{2}$

$$
x \neq y
$$

induction T-Abs
case T-App:
$\frac{\Gamma \leftarrow\left(x=\tau_{1}\right) \vdash e_{1}: \tau_{1} \rightarrow \tau_{2} \quad \Gamma \leftarrow\left(x=\tau_{1}\right) \vdash f: \tau_{1}}{\Gamma \leftarrow\left(x=\tau_{1}\right) \vdash e_{1} f: \tau_{2}}$
$\Gamma \vdash e_{1}[v / x]: \tau_{1} \rightarrow \tau_{2}$
induction
$\Gamma \vdash f[v / x]: \tau_{1}$
assumption
induction

$$
\begin{array}{lr}
\left(e_{1} f\right)[v / x]=\left(e_{1}[v / x]\right)(f[v / x]) & \text { def. subst. } \\
\Gamma \vdash\left(e_{1} f\right)[v / x]: \tau_{2} & \text { T-ABS }
\end{array}
$$

case T-Record:

$$
\begin{aligned}
& \frac{\left(\Gamma \leftrightarrow\left(x=\tau_{1}\right) \vdash e_{i} \tau_{i}\right)_{i \in 1 \ldots n}:}{\Gamma \leftrightarrow\left(x=\tau_{1}\right) \vdash(\overrightarrow{\ell=e}):(\overrightarrow{\ell: \tau})} \\
& (\overrightarrow{\ell=e})[v / x]=(\overrightarrow{\ell=e[v / x]}) \\
& \left(\Gamma \vdash e_{i}[v / x]: \tau_{i}\right)_{i \in 1 \ldots n} \\
& \Gamma \vdash(\overrightarrow{\ell=e})[v / x]:(\overrightarrow{\ell: \tau})
\end{aligned}
$$

assumption

$$
(\overrightarrow{\ell=e})[v / x]=(\overrightarrow{\ell=e[v / x]}) \quad \text { def. subst. }
$$

$$
\left(\Gamma \vdash e_{i}[v / x]: \tau_{i}\right)_{i \in 1 \ldots n} \quad \text { induction }
$$ T-Abs

case T-Const-Set:

$$
\begin{array}{lr}
\frac{\left(\Gamma \leftarrow\left(x=\tau_{1}\right) \vdash e_{i}: \tau\right)_{i \in 1 \ldots n}}{\Gamma \leftarrow\left(x=\tau_{1}\right) \vdash\{\vec{e}\}:\{\tau\}} & \text { assumption } \\
\{\vec{e}\}[v / x]=\{\overrightarrow{e[v / x]}\} & \text { def. subst. } \\
\left(\Gamma \vdash e_{i}[v / x]: \tau\right)_{i \in 1 \ldots n} & \text { induction } \\
\Gamma \vdash\{\vec{e}\}[v / x]:\{\tau\} & \text { T-Const-SET }
\end{array}
$$

case T-Lens:

$$
\begin{aligned}
& \frac{(S=\operatorname{Rel}(R, \text { true }, F)) \in \Phi \quad \bigcup \operatorname{names}(F) \subseteq \operatorname{dom}(R)}{\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash \text { lens } S \text { of } R \text { with } F: \text { lens of }(\{S\}, R, \text { true }, F)} \quad \text { assumption } \\
& \Gamma \vdash \text { lens } S \text { of } R \text { with } F: \text { lens of }(\{S\}, R, \text { true }, F)
\end{aligned} \quad \text { T-LENS } \quad ~ l
$$

case T-Select:

| $\Gamma \leftarrow\left(x=\tau_{1}\right) \vdash f:$ lens of $(\Sigma, R, P, F) \quad R \vdash Q:$ bool |  |
| :--- | ---: |
| $\quad F$ is in tree form $\quad P$ ignores outputs $(F)$ |  |
| $\Gamma \leftarrow\left(x=\tau_{1}\right) \vdash \operatorname{select}_{Q}$ from $f:$ lens of $(\Sigma, R, P \wedge Q, F)$ | assumption |
| $\left(\operatorname{select}_{Q}\right.$ from $\left.f\right)[v / x]=\operatorname{select}_{Q}$ from $f[v / x]$ | def. subst. |
| $\Gamma \vdash f[v / x]:$ lens of $(\Sigma, R, P, F)$ | induction |
| $\Gamma \vdash\left(\operatorname{select}_{Q}\right.$ from $\left.f\right)[v / x]:$ lens of $(\Sigma, R, P \wedge Q, F)$ | T-SELECT |

case T-Drop:

$$
\begin{aligned}
& F \equiv G \cup\{U \rightarrow \ell\} \quad \Gamma \leftarrow\left(x=\tau_{1}\right) \vdash f: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P, F\right) \\
& U \subseteq \operatorname{dom}(R) \quad \cdot \vdash w: A \\
& \mathbf{L J D}[R,(\ell: A)](P) \quad \operatorname{DV}[R,(\ell: A)](P,(\ell=w))
\end{aligned}
$$

$\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash$ drop $\ell$ determined by $(U, w)$ from $f:$ lens of $\left(\Sigma, R, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right)$
(drop $\ell$ determined by $(U, w)$ from $f$ ) $[v / x]$
$=\operatorname{drop} \ell$ determined by $(U, w)$ from $f[v / x]$ def. susbst.
$\Gamma \vdash f[v / x]:$ lens of $\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P, F\right) \quad$ induction
$\Gamma \vdash$ drop $\ell$ determined by $(U, w)$ from $f$ :lens of $\left(\Sigma, R, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right)$ T-Drop case T-Join-Templ:

$$
\begin{aligned}
& \Gamma \nleftarrow\left(x=\tau_{1}\right) \vdash e_{1}: \text { lens of }(\Sigma, R, P, F) \\
& \Gamma \leftrightarrow\left(x=\tau_{1}\right) \vdash e_{2} \text { : lens of }\left(\Delta, R^{\prime}, Q, G\right) \quad \Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash f: R \oplus R^{\prime} \rightarrow(\nwarrow|\uparrow| \nearrow) \\
& G \vDash \operatorname{dom}(R) \cap \operatorname{dom}\left(R^{\prime}\right) \rightarrow \operatorname{dom}\left(R^{\prime}\right) \quad F \text { is in tree form } \quad G \text { is in tree form } \\
& P \text { ignores outputs }(F) \quad Q \text { ignores outputs }(G) \quad \Sigma \cap \Delta=\varnothing \\
& \Gamma \leftrightarrow\left(x=\tau_{1}\right) \vdash \text { join }_{f} e_{1} \text { with } e_{2}: \text { lens of }\left(\Sigma \cup \Delta, R \oplus R^{\prime}, P \wedge Q, F \cup G\right) \\
& \text { assumption } \\
& \left(\text { join }_{f} e_{1} \text { with } e_{2}\right)[v / x]=\operatorname{join}_{f[v / x]} e_{1}[v / x] \text { with } e_{2}[v / x] \text { def. subst. } \\
& \Gamma \vdash e_{1}[v / x]: \text { lens of }(\Sigma, R, P, F) \quad \text { induction } \\
& \Gamma \vdash e_{2}[v / x]: \text { lens of }\left(\Delta, R^{\prime}, Q, G\right) \quad \text { induction } \\
& \Gamma \vdash f[v / x]: R \oplus R^{\prime} \rightarrow(\nwarrow|\uparrow| \nearrow) \quad \text { induction } \\
& \Gamma \vdash\left(\mathbf{j o i n}_{f} e_{1} \text { with } e_{2}\right)[v / x] \text { :lens of }\left(\Sigma \cup \Delta, R \oplus R^{\prime}, P \wedge Q, F \cup G\right) \text { T-Join-TempL }
\end{aligned}
$$ case T-Rename:

$\frac{\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash f: \text { lens of }(\Sigma, R \oplus(\ell: A), P, F)}{\Gamma \leftarrow\left(x=\tau_{1}\right) \vdash \text { rename }_{\ell / \ell^{\prime}} \text { from } f: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P\left[\ell / \ell^{\prime}\right], F\left[\ell / \ell^{\prime}\right]\right)}$
assumption
$\left(\right.$ rename $_{\ell / \ell^{\prime}}$ from $\left.f\right)[v / x]=\operatorname{rename}_{\ell / \ell^{\prime}}$ from $f[v / x]$ def. subst.
$\Gamma \vdash f:$ lens of $(\Sigma, R \oplus(\ell: A), P, F) \quad$ induction
$\Gamma \vdash\left(\right.$ rename $_{\ell / \ell^{\prime}}$ from $\left.f\right)[v / x]:$ lens of $\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P\left[\ell / \ell^{\prime}\right], F\left[\ell / \ell^{\prime}\right]\right)$
T-Rename
case T-GET:

$$
\begin{aligned}
& \Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash f: \text { lens of }(\Sigma, R, P, F) \\
& \Gamma \longleftrightarrow\left(x=\tau_{1}\right) \vdash \text { get } f:\{R\} \\
& \text { (get } f \text { ) }[v / x]=\text { get } f[v / x] \text { def. subst. } \\
& \Gamma \vdash f[v / x]: \text { lens of }(\Sigma, R, P, F) \text { induction } \\
& \Gamma \vdash(\text { get } f)[v / x]:\{R\}
\end{aligned}
$$

case T-Put:

| $\Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash e_{1}:$ lens of $(\Sigma, R, P, F) \quad \Gamma \longleftarrow\left(x=\tau_{1}\right) \vdash f:\{R\}$ | assumption |
| :---: | :---: |
| $\Gamma \leftarrow\left(x=\tau_{1}\right) \vdash$ put $e_{1}$ with $f$ : bool |  |
| (put $e_{1}$ with $f$ ) $[v / x]=$ put $e_{1}[v / x]$ with $f[v / x]$ | def. subst. |
| $\Gamma \vdash e_{1}[v / x]:$ lens of $(\Sigma, R, P, F)$ | induction |
| $\Gamma \vdash f[v / x]:\{R\}$ | induction |
| $\Gamma \vdash\left(\right.$ put $e_{1}$ with $\left.f\right)[v / x]$ : bool | T-Put |

Theorem 12 (Preservation). Given $\varphi: \Phi$, if $\cdot \vdash e: \tau$ and $e, \varphi \Downarrow v, \gamma$ then $\cdot \vdash v: \tau$ and $\gamma: \Phi$.

Proof.
$\varphi: \Phi$ assumption
$\cdot \vdash e: \tau \quad$ assumption (1)
$e, \varphi \Downarrow v, \gamma$
assumption (2)
Perform induction on $e, \varphi \Downarrow v, \gamma$
case E-Value
$\overline{v, \varphi \Downarrow v, \varphi}$ assumption
$\cdot \vdash v: \tau$
(1) $; e=v$
case E-App


| $\cdot \vdash v: \tau$ | induction |
| :--- | :--- |
| $\gamma: \Phi$ | induction |

case E-Record

$$
\begin{array}{lr}
\stackrel{\left(e_{i}, \varphi_{i} \Downarrow v_{i}, \varphi_{i+1}\right)_{i \in 1 \ldots n}}{(\overrightarrow{\ell=e}), \varphi_{1} \Downarrow(\overrightarrow{\ell=v}), \varphi_{n+1}} & \text { assumption } \\
\left(\Gamma \vdash e_{i}: \tau_{i}\right)_{i \in 1 \ldots n} & \text { (11); T-RECORD } \\
\left(\cdot \vdash v_{i}: \tau_{i}\right)_{i \in 1 \ldots n} & \text { induction } \\
\left(\varphi_{i+1}: \Phi\right)_{i \in 1 \ldots n} & \text { induction } \\
\cdot \vdash(\overrightarrow{\ell=v}): \tau & \text { T-RECORD }
\end{array}
$$

case E-Const-Set
E-Const-Set

$$
\frac{\left(e_{i}, \varphi_{i} \Downarrow v_{i}, \varphi_{i+1}\right)_{i \in 1 \ldots n} \quad \vec{w}=\operatorname{set} \text { of } v}{\{\vec{e}\}, \varphi_{1} \Downarrow\{\vec{w}\}, \varphi_{n+1}}
$$

assumption

$$
\left(\cdot \vdash e_{i}: \tau\right)_{i \in 1 \ldots n}
$$

(1); T-Const-Set

$$
\left(\cdot \vdash v_{i}: \tau\right)_{i \in 1 \ldots n}
$$ induction

$$
\left(\varphi_{i+1}: \Phi\right)_{i \in 1 \ldots n}
$$ induction

$$
\cdot \vdash\{\vec{v}\}:\{\tau\}
$$ T-Const-Set

case E-Lens-Select

$$
e, \varphi \Downarrow v, \gamma
$$

select $_{P}$ from $e, \varphi \Downarrow$ select $_{P}$ from $v, \gamma$

$$
\begin{array}{lr}
\Gamma \vdash e: \text { lens of }(\Sigma, R, P, F) \quad R \vdash Q: \text { bool } & \\
F \text { is in tree form } \quad P \text { ignores outputs }(F) & \text { 1 T-SELECT } \\
\hline \Gamma \vdash \operatorname{select}_{Q} \text { from } e: \text { lens of }(\Sigma, R, P \wedge Q, F) & \text { induction } \\
\Gamma \vdash v: \text { lens of }(\Sigma, R, P, F) & \text { induction } \\
\gamma: \Phi & \text { T-SELECT }
\end{array}
$$

case E-Join

$$
\frac{e_{1}, \varphi \Downarrow v_{1}, \varphi_{1} \quad e_{2}, \varphi_{1} \Downarrow v_{2}, \varphi_{2} \quad f, \varphi_{2} \Downarrow w, \gamma}{\operatorname{join}_{f} e_{1} \text { with } e_{2} \Downarrow \text { join }_{w} v_{1} \text { with } v_{2}}
$$

| $\Gamma \vdash e_{1}$ :lens of $(\Sigma, R, P, F) \quad \Gamma \vdash e_{2}:$ lens of $\left(\Delta, R^{\prime}, Q, G\right)$ |  |
| :---: | :---: |
| $\Gamma \vdash f: R \oplus R^{\prime} \rightarrow(\nwarrow\|\uparrow\| \nearrow) \quad G \vDash \operatorname{dom}(R) \cap \operatorname{dom}\left(R^{\prime}\right) \rightarrow \operatorname{dom}\left(R^{\prime}\right)$ |  |
| $F$ is in tree form $G$ is in tree form |  |
| $P$ ignores outputs( $F$ ) $\quad Q$ ignores outputs ( $G$ ) |  |
| $\Sigma \cap \Delta=\varnothing$ | (11) T-Join |
| $\cdot \vdash \operatorname{join}_{f} e_{1}$ with $e_{2}:$ lens of $\left(\Sigma \cup \Delta, R \oplus R^{\prime}, P \wedge Q, F \cup G\right)$ |  |
| $\cdot \vdash v_{1}$ : lens of ( $\left.\Sigma, R, P, F\right)$ | induction |
| $\varphi_{1}: \Phi$ | induction |
| $\cdot \vdash v_{2}:$ lens of $\left(\Delta, R^{\prime}, Q, G\right)$ | induction |
| $\varphi_{2}: \Phi$ | induction |
| $\cdot \vdash w: R \oplus R^{\prime} \rightarrow(\nwarrow\|\uparrow\| \nearrow)$ | induction |
| $\gamma: \Phi$ | induction |
| $\cdot \vdash \operatorname{join}_{w} v_{1}$ with $v_{2}:$ lens of $\left(\Sigma \cup \Delta, R \oplus R^{\prime}, P \wedge Q, F \cup G\right)$ | T-Join |
| ase E-Drop |  |

drop $\ell$ determined by $\left(U, v^{\prime}\right)$ from $e, \varphi \Downarrow$ drop $\ell$ determined by $\left(U, v^{\prime}\right)$ from $w, \gamma$
assumption

$$
\begin{gathered}
F \equiv G \cup\{U \rightarrow \ell\} \quad \vdash e: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P, F\right) \\
U \subseteq \operatorname{dom}(R) \quad \cdot \vdash v^{\prime}: A \\
\\
\mathbf{L J D}[R,(\ell: A)](P) \quad \operatorname{DV}[R,(\ell: A)](P,(\ell=v))
\end{gathered}
$$

$\Gamma \vdash \operatorname{drop} \ell$ determined by $\left(U, v^{\prime}\right)$ from $e:$ lens of $\left(\Sigma, R, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right)$
(11) T-Drop
$\begin{array}{lr}\cdot \vdash w: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P, F\right) & \text { induction } \\ \gamma: \Phi & \text { induction } \\ \cdot \vdash \text { drop } \ell \text { determined by }\left(U, v^{\prime}\right) \text { from } w: \text { lens of }\left(\Sigma, R, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right) \text { T-DROP }\end{array}$ case E-Rename

| $f, \varphi \Downarrow w, \gamma$ | assumption |
| :---: | :---: |
| rename $_{\ell / \ell^{\prime}}$ from $f, \varphi \Downarrow$ rename $_{\ell / \ell^{\prime}}$ from $w, \gamma$ |  |
| $\cdot \vdash f:$ lens of $(\Sigma, R \oplus(\ell: A), P, F)$ | (1]; T-Rename |
| $\cdot \vdash \operatorname{rename}_{\ell / \ell^{\prime}}$ from $f:$ lens of $\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P\left[\ell / \ell^{\prime}\right], F\left[\ell / \ell^{\prime}\right]\right)$ |  |
| $\cdot \vdash w:$ lens of $(\Sigma, R \oplus(\ell: A), P, F)$ | induction |
| $\gamma: \Phi$ | induction |

$\cdot \vdash$ rename $_{\ell / \ell^{\prime}}$ from $w:$ lens of $\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P\left[\ell / \ell^{\prime}\right], F\left[\ell / \ell^{\prime}\right]\right) \quad$ T-Rename
case E-Get

$$
\begin{array}{lr}
\frac{e^{\prime}, \varphi \Downarrow v^{\prime}, \gamma \quad w=g e t_{v^{\prime}}(\gamma)}{\text { get } e, \varphi \Downarrow w, \gamma} & \text { assumption } \\
\frac{\cdot \vdash e^{\prime}: \text { lens of }(\Sigma, R, P, F)}{\cdot \vdash \text { get } e^{\prime}:\{R\}} & \text { (11; T-GET } \\
\cdot \vdash v^{\prime}: \text { lens of }(\Sigma, R, P, F) & \text { induction } \\
\gamma: \Phi & \text { induction } \\
\left(v^{\prime}\right)=\Sigma / I / S & \text { Theorem 11 } \\
\operatorname{sort}(S)=(\operatorname{dom}(R), \operatorname{set}(P, R), F) & \text { Theorem 11 } \\
\cdot \vdash w:\{R\} & I \text { well-behaved }
\end{array}
$$

case E-Put-Sat
$e, \varphi \Downarrow v^{\prime}, \varphi_{1} \quad f, \varphi_{1} \Downarrow w, \varphi_{2} \quad \cdot \vdash v^{\prime}:$ lens of $(\Sigma, S, P, F)$
$w \models F \quad \forall r \in w . \operatorname{sat}(P, r) \quad \gamma=p u t_{v}\left(\varphi_{2}, w\right)$
put $e$ with $f, \varphi \Downarrow$ true, $\gamma$
$\Gamma \vdash e:$ lens of $(\Sigma, R, P, F) \quad \Gamma \vdash f:\{R\}$
$\Gamma \vdash$ put $e$ with $f$ : bool
$\varphi_{1}: \Phi$
$\varphi_{2}: \Phi$
$\cdot \vdash w:\{R\}$
$w: \operatorname{Rel}(R, P, F)$
$\cdot \vdash$ true:bool
$\gamma: \Phi$
case E-Put-Unsat
$e, \varphi \Downarrow v, \varphi_{1} \quad f, \varphi_{1} \Downarrow w, \gamma \quad v:$ lens of $(\Sigma, S, P, F)$
$w \nvdash F$ or $\exists r \in w$. $\neg \operatorname{sat}(P, r)$
put $e$ with $f, \varphi \Downarrow$ false, $\gamma$
$\Gamma \vdash e:$ lens of $(\Sigma, R, P, F) \quad \Gamma \vdash f:\{R\}$
$\Gamma \vdash$ put $e$ with $f$ : bool
$\varphi_{1}: \Phi$
$\gamma: \Phi$
assumption
(1); T-Put (3)
(3); induction (4)
(3. 4); induction (5)
(3) 4); induction
def. $\operatorname{Rel}(R, P, F)(6)$
(6); T-Const
(5); Lemma 66
assumption
(11; T-Put (7)
(7); induction (8)
(7. 8); induction

## D. 2 Naive Lens Semantics

Chapter 5 shows how relational lenses can be integrated into the type system of a functional language. The chapter abstracts over the actual lens semantics. For any lens value $v$ we require an implementation for the $\operatorname{get}_{v}(\varphi)$ and $\operatorname{put}_{v}(\varphi, M)$ functions, which should query and update the database value $\varphi$. We first provide definitions for get and put for each lens primitive. We rely on the non-incremental semantics, and all of these functions are simple adaptions from the relational lens semantics shown in Chapter 2 We later show that the round-tripping properties required in 5 apply to these lenses.

Definition 32. Given the value $v=$ lens $S$ of $R$ with $F$ such that $\cdot \vdash v:$ lens of $(\Sigma, R, Q, F)$, define the functions get ${ }_{v}$ and $p u t_{v}$ as follows:

$$
\begin{aligned}
\operatorname{get}_{v}(\varphi) & =\varphi(S) \\
\operatorname{put}_{v}(\varphi, w) & =\varphi_{\backslash S} \leftarrow(S=w)
\end{aligned}
$$

Definition 33. Given the value $v=$ select $_{w}$ from $P$ such that $\cdot \vdash v:$ lens of $(\Sigma, R, Q, F)$, define the functions get $v_{v}$ and put ${ }_{v}$ as follows:

$$
\begin{aligned}
\operatorname{get}_{v}(\varphi)= & \sigma_{P}\left(\operatorname{get}_{w}(\varphi)\right) \\
\operatorname{put}_{v}(\varphi, M)= & \text { let } N=\operatorname{get}_{w}(\varphi) \text { in } \\
& \text { let } M_{0}=\operatorname{merge}_{F}\left(\sigma_{\neg P}(M), \varphi\right) \text { in } \\
& \text { let } N_{\#}=\sigma_{P}\left(M_{0}\right)-N \text { in } \\
& \operatorname{put}_{w}\left(\varphi, M_{0}-N_{\#}\right)
\end{aligned}
$$

Definition 34. Given the value $v=$ drop $\ell$ determined by $\left(U, v^{\prime}\right)$ from $w$ such that $\cdot \vdash v: l e n s$ of $(\Sigma, R, Q, F)$, define the functions get ${ }_{v}$ and put $_{v}$ as follows:

$$
\begin{aligned}
& \operatorname{get}_{v}(\varphi)=\pi_{U-\ell}\left(\operatorname{get}_{w}(\varphi)\right) \\
& \operatorname{put}_{v}(\varphi, M)=\operatorname{let} N=\operatorname{get}_{w}(\varphi) \text { in } \\
& \text { let } M^{\prime}=N \bowtie\left\{\left(\ell=v^{\prime}\right)\right\} \text { in } \\
& \text { revise }_{U \rightarrow \ell}\left(M^{\prime}, M\right)
\end{aligned}
$$

Definition 35. Given the value $v=\boldsymbol{j o i n}_{w} v_{1}$ with $v_{2}$ such that $\cdot \vdash v:$ lens $\boldsymbol{o f}(\Sigma, R, Q, F)$, define the functions get $_{v}$ and put $_{v}$ as follows:

$$
\begin{aligned}
\operatorname{get}_{v}(\varphi)= & \operatorname{get}_{v_{1}}(\varphi) \bowtie \operatorname{get}_{v_{2}}(\varphi) \\
\operatorname{put}_{v}(\varphi, O)= & \text { let } M=\operatorname{get}_{v_{1}}(\varphi) \text { in } \\
& \text { let } N=\operatorname{get}_{v_{2}}(\varphi) \text { in } \\
& \text { let } M_{0}=\operatorname{merge}_{F}\left(M, \pi_{U}(O)\right) \text { in } \\
& \text { let } N_{0}=\operatorname{merge}_{G}\left(N, \pi_{V}(O)\right) \text { in } \\
& \text { let } L=\left(M_{0} \bowtie N_{0}\right)-O \text { in } \\
& \text { let } L_{a}=L \bowtie \pi_{U \cap V}(O) \text { in } \\
& \text { let } L_{l}=L-L_{a} \text { in } \\
& \text { let } M^{\prime}=M_{0}-\pi_{U}\left(L_{l} \cup \sigma_{P_{d}}\left(L_{a}\right)\right) \text { in } \\
& \text { let } N^{\prime}=N_{0}-\pi_{V}\left(\sigma_{Q_{d}}\left(L_{a}\right)\right) \text { in } \\
& \operatorname{put}_{v_{2}}\left(p u t_{v_{1}}\left(\varphi, M^{\prime}\right), N^{\prime}\right)
\end{aligned}
$$

$P_{d}$ and $Q_{d}$ are defined by $w$ according to Definition 28.
Definition 36. Given the value $v=$ rename $_{\ell / \ell^{\prime}}$ from $w$ such that $\cdot \vdash v: l e n s$ of $(\Sigma, R, Q, F)$ is define the functions get and put as follows:

$$
\begin{aligned}
\operatorname{get}_{v}(\varphi) & =\rho_{\ell / \ell^{\prime}}\left(\operatorname{get}_{w}(\varphi)\right) \\
\operatorname{put}_{v}(\varphi, M) & =\operatorname{put}_{w}\left(\varphi, \rho_{\ell^{\prime} / \ell}(M)\right)
\end{aligned}
$$

To show our lenses satisfy the required lemmas, we rely on the fact that we can rewrite the semantics of each lens using the definitions from Chapter 2. For example, given a select lens $v=\operatorname{select}_{P}$ from $w$, it is safe to construct the lens $I=\operatorname{select}_{P}$. The definitions for get and put can then be written as:

- $\operatorname{get}_{v}(\varphi)=\operatorname{get}_{I}\left(\operatorname{get}_{w}(\varphi)\right)$
- $\operatorname{put}_{v}(\varphi, M)=\operatorname{put}_{w}\left(\varphi, \operatorname{put}_{I}\left(\operatorname{get}_{w}(\varphi), M\right)\right)$

We first show that the put and get functions are total. For this we rely on the underlying lens to be total and well-typed (see Appendix A.1).

Lemma 65 (Get Total). If $\varphi: \Phi$ and $\cdot \vdash v$ :lens of $(\Sigma, R, P, F)$ then $\operatorname{get}_{v}(\varphi)=M$ and $M: \operatorname{Rel}(R, P, F)$.

Proof.
$\varphi: \Phi$
assumption
$M: \operatorname{Rel}(R, P, F)$
assumption
$\cdot \vdash v:$ lens of $(\Sigma, R, Q, F)$ assumption
perform induction on $\cdot \vdash v$ : lens of $(\Sigma, R, Q, F)$
case T-Lens-Prim

$$
\frac{(S=\operatorname{Rel}(R, \text { true }, F)) \in \Phi \quad \bigcup \operatorname{names}(F) \subseteq \operatorname{dom}(R)}{\Gamma \vdash \text { lens } S \text { of } R \text { with } F: \text { lens of }(\{S\}, R, \text { true }, F)} \quad \quad \text { assumption }
$$

$\exists N: \operatorname{Rel}(R$, true,$F) . \varphi(S)=N$

$$
\operatorname{get}_{v}(\varphi)=N: \operatorname{Rel}(R, \text { true }, F)
$$

case T-Select

| $\Gamma \vdash e:$ lens of $(\Sigma, R, P, F) \quad R \vdash Q$ : bool |  |
| :---: | :---: |
| $F$ is in tree form $\quad P$ ignores outputs ( $F$ ) |  |
| $\Gamma \vdash \operatorname{select}_{Q}$ from $e:$ lens of $(\Sigma, R, P \wedge Q, F)$ |  |
| $v=\operatorname{select}_{Q}$ from $w$ | assumption |
| $I=\operatorname{select}_{Q}$ | define |
| $M=\operatorname{get}_{w}(\varphi): \operatorname{Rel}(R, P, F)$ | induction |
| $g e t_{v}(\varphi)$ |  |
| $=\operatorname{get}_{I}\left(\operatorname{get}_{w}(\varphi)\right)$ | def. get $_{v}(\cdot)$ |
| $=\operatorname{get}_{I}(M): \operatorname{Rel}(R, P \wedge Q, F)$ | get $_{I}(M)$ total |

case T-Drop

$$
\begin{aligned}
& F \equiv G \cup\{U \rightarrow \ell\} \quad \Gamma \vdash e: \text { lens of }\left(\Sigma, R^{\prime} \oplus\left(\ell^{\prime}: A\right), P, F\right) \\
& U \subseteq \operatorname{dom}(R) \quad \cdot \vdash v^{\prime}: A \\
& \operatorname{LJD}\left[R^{\prime},(\ell: A)\right](P) \quad \mathbf{D V}\left[R^{\prime},(\ell: A)\right]\left(P,\left(\ell=v^{\prime}\right)\right) \\
& \Gamma \vdash \operatorname{drop} \ell \text { determined by }\left(U, v^{\prime}\right) \text { from } e \text { lens of }\left(\Sigma, R^{\prime}, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right) \\
& \text { assumption } \\
& v=\operatorname{drop} \ell \text { determined by }\left(U, v^{\prime}\right) \text { from } w \\
& \text { assumption } \\
& I=\operatorname{drop} \ell \text { determined by }\left(U, v^{\prime}\right) \\
& N=\operatorname{get}_{w}(\varphi): \operatorname{Rel}(R, P, F) \\
& \text { define } \\
& \text { induction } \\
& \operatorname{get}_{v}(\varphi)=\operatorname{get}_{I}\left(\operatorname{get}_{w}(\varphi)\right): \operatorname{Rel}\left(R^{\prime}, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right) \\
& \operatorname{get}_{I}(\cdot) \text { total }
\end{aligned}
$$

case T-Join

$$
\begin{aligned}
& \Gamma \vdash v_{1}: \text { lens of }\left(\Sigma, R_{1}, P_{1}, F_{1}\right) \\
& \Gamma \vdash v_{2} \text { :lens of }\left(\Delta, R_{2}, Q, G\right) \quad \Gamma \vdash f: R_{1} \cup R_{2} \rightarrow(\nwarrow|\uparrow| \nearrow) \\
& G \vDash \operatorname{dom}\left(R_{1}\right) \cap \operatorname{dom}\left(R_{2}\right) \rightarrow \operatorname{dom}\left(R_{2}\right) \quad F \text { is in tree form } \quad G \text { is in tree form } \\
& P \text { ignores outputs }(F) \quad Q \text { ignores outputs }(G) \quad \Sigma \cap \Delta=\varnothing \\
& \Gamma \vdash \operatorname{join}_{f} e_{1} \text { with } e_{2}: \text { lens of }\left(\Sigma \cup \Delta, R \cup R^{\prime}, P \wedge Q, F_{1} \cup G\right) \\
& v=\boldsymbol{j o i n}_{w} v_{1} \text { with } v_{2} \quad \text { assumption } \\
& I=\operatorname{join}_{P_{d}, Q_{d}} \\
& \operatorname{get}_{v_{1}}(\varphi): \operatorname{Rel}\left(R_{1}, P_{1}, F_{1}\right) \quad \text { induction } \\
& \operatorname{get}_{v_{2}}(\varphi): \operatorname{Rel}\left(R_{2}, Q, G\right) \quad \text { induction } \\
& \operatorname{get}_{v}(\varphi)=\operatorname{get}_{I}\left(\operatorname{get}_{v_{1}}(\varphi), \operatorname{get}_{v_{2}}(\varphi)\right): \operatorname{Rel}\left(R_{1} \cup R_{2}, P_{1} \wedge Q, F_{1} \cup G\right) \quad g e t_{I} \text { total }
\end{aligned}
$$

case T-Rename

| $\frac{\Gamma \vdash e: \operatorname{lens} \operatorname{of}\left(\Sigma, R^{\prime} \oplus(\ell: A), P, F\right)}{}$ |  |
| :--- | ---: |
| $\Gamma \vdash \operatorname{rename}_{\ell / \ell^{\prime}}$ from $e: \operatorname{lens}$ of $\left(\Sigma, R^{\prime} \oplus\left(\ell^{\prime}: A\right), P\left[\ell / \ell^{\prime}\right], F\left[\ell / \ell^{\prime}\right]\right)$ |  |
| $v=\operatorname{rename}_{\ell / \ell^{\prime}}$ from $w$ | assumption |
| $I=\operatorname{rename}_{\ell / \ell^{\prime}}$ | define |
| $\operatorname{get}_{w}(\varphi): \operatorname{Rel}(R, P, F)^{\operatorname{get}_{v}(\varphi)=\operatorname{get}_{I}\left(\operatorname{get}_{w}(\varphi)\right): \operatorname{Rel}\left(R^{\prime}, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right)}$ | induction |
| get $_{I}$ total |  |

Lemma 66 (Put Total). Given $\varphi: \Phi$, if $M: \operatorname{Rel}(R, P, F)$ and $\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$ then $\operatorname{put}_{v}(\varphi, M)=\gamma$ and $\gamma: \Phi$.

Proof.
$\varphi: \Phi$
assumption (1)
$M: \operatorname{Rel}(R, P, F)$
assumption
$\cdot \vdash v:$ lens of $(\Sigma, R, Q, F)$
assumption
perform induction on $\cdot \vdash v:$ lens of $(\Sigma, R, Q, F)$
case T-Lens-Prim

$$
\frac{(S=\operatorname{Rel}(R, \text { true }, F)) \in \Phi \quad \bigcup \operatorname{names}(F) \subseteq \operatorname{dom}(R)}{\Gamma \vdash \text { lens } S \text { of } R \text { with } F: \text { lens of }(\{S\}, R, \text { true }, F)}
$$

$$
\begin{equation*}
\exists N: \operatorname{Rel}(R, \text { true }, F) \cdot \varphi(S)=N \tag{1}
\end{equation*}
$$

$$
\begin{aligned}
& M: \operatorname{Rel}(R, \text { true }, F) \\
& p u t_{v}(\varphi, M)=\varphi_{\backslash S} \leftarrow(S=M) \\
& \varphi_{\backslash S} \leftarrow(S=M): \Phi
\end{aligned}
$$

$$
\text { def. } P(2)
$$

$$
\operatorname{put}_{v}(\varphi, M): \Phi \quad \text { def. } \cdot \leftarrow \cdot
$$

case T-Select
$\Gamma \vdash e:$ lens of $(\Sigma, R, P, F) \quad R \vdash Q$ : bool
$F$ is in tree form $\quad P$ ignores outputs $(F)$
$\Gamma \vdash \operatorname{select}_{Q}$ from $e$ lens of $(\Sigma, R, P \wedge Q, F)$
$v=\operatorname{select}_{Q}$ from $w \quad$ assumption
$I=\operatorname{select}_{Q}$
$N=\operatorname{get}_{w}(\varphi): \operatorname{Rel}(R, P, F)$
define
Lemma 65
$\operatorname{put}_{I}(N, M): \operatorname{Rel}(R, P, F) \quad$ put $_{I}$ total
$\operatorname{put}_{v}(\varphi, M)=\operatorname{put}_{w}\left(\varphi\right.$, put $_{I}\left(\right.$ get $\left.\left._{w}(\varphi), M\right)\right): \Phi \quad$ def. $p u t_{v} ;$ induction
case T-Drop

$$
\begin{aligned}
& F \equiv G \cup\{U \rightarrow \ell\} \quad \Gamma \vdash e: \text { lens of }\left(\Sigma, R \oplus\left(\ell^{\prime}: A\right), P, F\right) \\
& U \subseteq \operatorname{dom}(R) \quad \cdot \vdash v: A \\
& \mathbf{L J D}[R,(\ell: A)](P) \quad \mathbf{D V}[R,(\ell: A)](P,(\ell=v)) \\
& \Gamma \vdash \text { drop } \ell \text { determined by }(U, v) \text { from } e: \text { lens of }\left(\Sigma, R, \llbracket P \rrbracket_{R_{1}, R_{2}}, G\right) \\
& \text { assumption } \\
& v=\operatorname{drop} \ell \text { determined by }\left(U, v^{\prime}\right) \text { from } w \\
& I=\text { drop } \ell \text { determined by }\left(U, v^{\prime}\right) \\
& N=\operatorname{get}_{v}(\varphi): \operatorname{Rel}(R, P, F) \\
& \text { Lemma } 65 \\
& p u t_{I}(M, N): \operatorname{Rel}(R, P, F) \\
& \text { put }_{\text {I }} \text { total } \\
& \operatorname{put}_{v}(\varphi, M)=\operatorname{put}_{w}\left(\varphi, \operatorname{put}_{I}\left(\operatorname{get}_{w}(\varphi), M\right)\right): \Phi \\
& \text { assumption } \\
& \text { define } \\
& \text { def. } p u t_{v} \text {; induction } \\
& \text { case T-Join } \\
& \Gamma \vdash v_{1}: \text { lens of }\left(\Sigma, R_{1}, P_{1}, F_{1}\right) \\
& \Gamma \vdash v_{2} \text { :lens of }\left(\Delta, R_{2}, Q, G\right) \quad \Gamma \vdash f: R_{1} \cup R_{2} \rightarrow(\nwarrow|\uparrow| \nearrow) \\
& G \vDash \operatorname{dom}\left(R_{1}\right) \cap \operatorname{dom}\left(R_{2}\right) \rightarrow \operatorname{dom}\left(R_{2}\right) \quad F \text { is in tree form } \quad G \text { is in tree form } \\
& P \text { ignores outputs }(F) \quad Q \text { ignores outputs }(G) \quad \Sigma \cap \Delta=\varnothing \\
& \Gamma \vdash \mathbf{j o i n}_{f} e_{1} \text { with } e_{2}: \text { lens of }\left(\Sigma \cup \Delta, R \cup R^{\prime}, P \wedge Q, F_{1} \cup G\right)
\end{aligned}
$$

$$
\begin{aligned}
& I=\operatorname{join}_{P_{d}, Q_{d}} \\
& \text { define } \\
& \text { * } P_{d} \text { and } Q_{d} \text { defined from } w \text { according to Definition } 28 \\
& \operatorname{get}_{v_{1}}(\varphi): \operatorname{Rel}\left(R_{1}, P_{1}, F_{1}\right) \\
& \operatorname{get}_{v_{2}}(\varphi): \operatorname{Rel}\left(R_{2}, Q, G\right) \\
& \left(M^{\prime}, N^{\prime}\right)=\operatorname{put}_{I}\left(\left(\operatorname{get}_{v_{1}}(\varphi), \operatorname{get}_{v_{2}}(\varphi)\right), M\right) \\
& M^{\prime}: \operatorname{Rel}\left(S_{1}, P_{1}, F_{1}\right) \quad \text { put }_{I} \text { total } \\
& N^{\prime}: \operatorname{Rel}\left(S_{2}, Q, G\right) \quad \text { put }_{I} \text { total } \\
& \text { put }_{v_{1}}\left(\varphi, M^{\prime}\right): \Phi \quad \text { induction (3) } \\
& \operatorname{put}_{v}(\varphi, M)=\text { put }_{v_{2}}\left(\text { put }_{v_{1}}\left(\varphi, M^{\prime}\right), N^{\prime}\right): \Phi \\
& \text { Lemma } 65 \\
& \text { Lemma } 65 \\
& \text { define } \\
& \text { put }_{\text {I }} \text { total } \\
& \text { put }_{\text {I }} \text { total } \\
& \text { induction (3) } \\
& \text { (3); induction }
\end{aligned}
$$

case T-Rename


For the round-tripping properties we rely on the lens $I$ being well-behaved. This means we can use the following equivalences:

- $\operatorname{get}_{I}\left(\operatorname{put}_{I}(M, N)\right)=N$
- $\operatorname{put}_{I}\left(M, \operatorname{get}_{I}(M)\right)=M$

Lemma 68 (GetPut). If $\varphi: \Phi$ and $\cdot \vdash v$ : lens of $(\Sigma, R, P, F)$ then $\operatorname{put}_{v}\left(\varphi, \operatorname{get}_{v}(\varphi)\right)=\varphi$.

Proof.
$\cdot \vdash v:$ lens of $(\Sigma, R, Q, F)$ assumption
perform induction on $\cdot \vdash v:$ lens of $(\Sigma, R, Q, F)$
case T-Lens-Prim
$v=$ lens $S$ of $R$ with $F$
assumption

$$
\begin{aligned}
& p u t_{v}(\varphi, \operatorname{get}(\varphi)) \\
& \quad=\varphi \backslash S \leftarrow \varphi \\
& \quad=\varphi
\end{aligned}
$$

$$
=\varphi_{\backslash S} \longleftarrow \varphi(S) \quad \text { def. get, put }
$$

$$
\text { def. } \cdot \leftarrow
$$

case T-SELECT
$v=\operatorname{select}_{P}$ from $v \quad$ assumption

$$
\begin{aligned}
& I=\operatorname{select}_{P} \\
& \begin{aligned}
& M=\operatorname{get}_{w}(\varphi) \\
& p u t_{v}\left(\varphi, \text { get }_{v}(\varphi)\right) \\
&=\text { put }_{w}\left(\varphi, \text { put }_{I}\left(M, \text { get }_{I}(M)\right)\right) \\
&=\text { put }_{w}(\varphi, M) \\
&=\text { put }_{w}\left(\varphi, \text { get }_{w}(\varphi)\right) \\
&=\varphi
\end{aligned}
\end{aligned}
$$

define define
def. $p u t_{v}$
PutGet for $I$
def. $M$ induction
case T-Drop

$$
v=\operatorname{drop} \ell \text { determined by }\left(U, v^{\prime}\right) \text { from } w
$$

assumption
$I=\operatorname{drop} \ell$ determined by $(U, v)$
$M=g e t_{w}(\varphi)$
$\operatorname{put}_{v}\left(\varphi, \operatorname{get}_{v}(\varphi)\right)$
$=p u t_{w}\left(\varphi, \operatorname{put}_{I}\left(M, \operatorname{get}_{I}(M)\right)\right)$
$=p u t_{w}(\varphi, M)$
$=p u t_{w}\left(\varphi, \operatorname{get}_{w}(\varphi)\right)$
$=\varphi$
case T-Join

$$
\begin{array}{lr}
v=\text { join }_{w} v_{1} \text { with } v_{2} & \text { assumption } \\
I=\text { join }_{P_{d}, Q_{d}} & \text { define } \\
\quad * P_{d} \text { and } Q_{d} \text { defined from } w \text { according to Definition } 28 & \\
M=\text { get }_{v_{1}}(\varphi) & \text { define } \\
N=\text { get }_{v_{2}}(\varphi) & \text { define } \\
\left(M^{\prime}, N^{\prime}\right)=\text { put }_{I}\left(\text { get }_{I}(M, N)\right) & \text { define } \\
\quad=(M, N) & \text { PUTGET for } I(1) \\
(M, N)=\text { get }_{w}(\varphi) & \text { define }
\end{array}
$$

$$
\begin{align*}
& =\text { put }_{v_{2}}\left(\text { put }_{v_{1}}\left(\varphi, M^{\prime}\right), N^{\prime}\right) \\
& =\text { put }_{v_{2}}\left(\text { put }_{v_{1}}(\varphi, M), N\right)  \tag{1}\\
& =\text { put }_{v_{2}}(\varphi, N) \\
& =\varphi
\end{align*}
$$

def. $p u t_{v}$
induction
induction
case T-REname

$$
\begin{array}{rlr}
v & =\operatorname{rename}_{\ell / \ell^{\prime}} \text { from } w & \text { assumption } \\
I & =\operatorname{rename}_{\ell / \ell^{\prime}} & \text { define } \\
M & =\text { get }_{w}(\varphi) & \text { define } \\
p^{\prime} t_{v}\left(\varphi, \text { get }_{v}(\varphi)\right) & \\
& =\text { put }_{w}\left(\varphi, \text { put }_{I}\left(M, \text { get }_{I}(M)\right)\right) & \text { def. } p u t_{v} \\
& =\operatorname{put}_{w}(\varphi, M) & \text { PUTGET for } I \\
& =\text { put }_{w}\left(\varphi, \text { get }_{w}(\varphi)\right) & \text { def. } M \\
& =\varphi & \text { induction }
\end{array}
$$

Lemma 94. If $\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$ and $\cdot \vdash w:$ lens of $\left(\Delta, R^{\prime}, Q, G\right)$ and $\Sigma$ and $\Delta$ are disjoint, then $\operatorname{get}_{v}\left(\operatorname{put}_{w}(\varphi, M)\right)=\operatorname{get}_{v}(\varphi)$.

Proof. The lens $w$ only ever modifies tables in $\Delta$, so any tables referenced by $\Sigma$ which may affect the result are unchanged.

Lemma 69 (PutGet). If $\varphi: \Phi$ and $\cdot \vdash v:$ lens of $(\Sigma, R, P, F)$ and $M: \operatorname{Rel}(R, P, F)$ then $\operatorname{get}_{v}\left(\operatorname{put}_{v}(\varphi, M)\right)=M$.

Proof.
$\cdot \vdash v:$ lens of $(\Sigma, R, Q, F)$
assumption
perform induction on $\cdot \vdash v:$ lens of $(\Sigma, R, Q, F)$
case T-LEns-PRIm
$v=$ lens $S$ of $R$ with $F$
assumption
$\operatorname{get}_{v}\left(p u t_{v}(\varphi, M)\right)$
$=\operatorname{get}\left(\varphi_{\backslash S} \longleftarrow M\right)(S)$ def. get, put
$=M \quad$ def. $\cdot \leftarrow$.
case T-SELECT

$$
\begin{aligned}
& v=\operatorname{select}_{P} \text { from } v \\
& \begin{aligned}
I & =\operatorname{select}_{P} \\
& g e t_{v}\left(\text { put }_{v}(\varphi, M)\right) \\
& =\operatorname{get}_{I}\left(\operatorname{get}_{w}\left(p u t_{v}(\varphi, M)\right)\right) \\
& =\operatorname{get}_{I}\left(g e t_{w}\left(p u t_{w}\left(\varphi, p u t_{I}\left(g e t_{w}(\varphi), M\right)\right)\right)\right) \\
& =\operatorname{get}_{I}\left(p u t_{I}\left(g e t_{w}(\varphi), M\right)\right) \\
& =M
\end{aligned}
\end{aligned}
$$

assumption
def. get $_{v}$ def. $p u t_{v}$ induction PutGet for $I$

$$
I=\operatorname{select}_{P} \quad \text { define }
$$

case T-Drop
$v=\operatorname{drop} \ell$ determined by $\left(U, v^{\prime}\right)$ from $w$
assumption
define
def. get $_{v}$
def. $p u t_{v}$ induction PutGet for $I$
case T-Join

$$
\begin{aligned}
& v=\operatorname{join}_{w} v_{1} \text { with } v_{2} \\
& I=\operatorname{join}_{P_{d}, Q_{d}} \\
& \text { * } P_{d} \text { and } Q_{d} \text { defined from } w \text { according to Definition } 28 \\
& \left(M^{\prime}, N^{\prime}\right)=\operatorname{put}_{I}\left(\left(\operatorname{get}_{v_{1}}(\varphi), \operatorname{get}_{v_{2}}(\varphi)\right), O\right) \\
& p u t_{v}(\varphi, O) \\
& =p u t_{v_{2}}\left(\text { put }_{v_{1}}\left(\varphi, M^{\prime}\right), N^{\prime}\right) \\
& \operatorname{get}_{v_{1}}\left(p u t_{v}(\varphi, O)\right) \\
& =\text { get }_{v_{1}}\left(\text { put }_{v_{2}}\left(\text { put }_{v_{1}}\left(\varphi, M^{\prime}\right), N^{\prime}\right)\right) \\
& =\operatorname{get}_{v_{1}}\left(p u t_{v_{1}}\left(\varphi, M^{\prime}\right)\right) \\
& =M^{\prime} \\
& \operatorname{get}_{v_{2}}\left(\text { put }_{v}(\varphi, O)\right) \\
& =\text { get }_{v_{2}}\left(\text { put }_{v_{2}}\left(\text { put }_{v_{1}}\left(\varphi, M^{\prime}\right), N^{\prime}\right)\right) \\
& =N^{\prime} \\
& \operatorname{get}_{v}\left(\operatorname{put}_{v}(\varphi, O)\right) \\
& =\operatorname{get}_{I}\left(\operatorname{get}_{v_{1}}\left(\operatorname{put}_{v}(\varphi, O)\right), \operatorname{get}_{v_{2}}\left(\text { put }_{v}(\varphi, O)\right)\right) \\
& \text { assumption } \\
& \text { define } \\
& \text { define } \\
& \text { def. } p u t_{v} \\
& \text { def. } p u t_{v} \\
& \text { Lemma } 94 \\
& \text { induction } \\
& \text { def. } p u t_{v} \\
& \text { induction } \\
& \text { def. } g e t_{v}
\end{aligned}
$$

$$
\begin{aligned}
& =\operatorname{get}_{I}\left(\operatorname{get}_{w}\left(\operatorname{put}_{w}\left(\varphi, \operatorname{put}_{I}\left(\operatorname{get}_{w}(\varphi), M\right)\right)\right)\right) \\
& =\operatorname{get}_{I}\left(\operatorname{put}_{I}\left(\operatorname{get}_{w}(\varphi), M\right)\right) \\
& =M
\end{aligned}
$$

def. $p u t_{v}$ induction PutGet for $I$
case T-Rename
$v=\operatorname{rename}_{\ell / \ell^{\prime}}$ from $w \quad$ assumption
$I=$ rename $_{\ell / \ell^{\prime}} \quad$ define

$$
\operatorname{get}_{v}\left(p u t_{v}(\varphi, M)\right)
$$

$$
=\operatorname{get}_{I}\left(\operatorname{get}_{w}\left(p u t_{v}(\varphi, M)\right)\right) \quad \text { def. } \text { get }_{v}
$$

$$
=\operatorname{get}_{I}\left(\operatorname{get}_{w}\left(p u t_{w}\left(\varphi, p u t_{I}\left(\operatorname{get}_{w}(\varphi), M\right)\right)\right)\right)
$$

$$
=\operatorname{get}_{I}\left(\operatorname{put}_{I}\left(\operatorname{get}_{w}(\varphi), M\right)\right)
$$

$$
=M
$$

def. $p u t_{v}$
induction
PutGet for $I$

## D. 3 Empty Drop Lens

In this section we would like to show that constructing a drop lens for any lens such that $\operatorname{set}(P, R)=\varnothing$ is still sound.

The drop lens typing rules from Bohannon et al. [12] does not permit any lens where the predicate $P=\varnothing$ (where $P$ is a set predicate), because it does not satisfy the default value constraint $\{A=a\} \in \pi_{A}(P)$. This lens, even if not very useful, is still well-behaved as long as the resulting predicate set is also empty. Instead the only permitted view is the $\varnothing$ value. Consider a lens with the following semantics:

$$
\begin{aligned}
\operatorname{get}(M) & =\varnothing \\
\operatorname{put}(\varnothing, M) & =M
\end{aligned}
$$

It is easy to show that this lens is well-behaved:

$$
\begin{aligned}
\operatorname{get}(p u t(\varnothing, M)) & =\varnothing \\
\operatorname{put}(\operatorname{get}(M), M) & =M
\end{aligned}
$$

For the typing rules to remain well-behaved, it is necessary to show that $\llbracket P \rrbracket_{R_{1}, R_{2}}$ preserves predicates that are empty. We start by specifying the inductive rules $\operatorname{NoSat}\left[R_{1}, R_{2}\right](P, r)$,
which specify that the record $r$ : $R_{1}$ causes an evaluation of $P$ to yield false.

NoSat-1
$\frac{R_{1} \vdash P \text { : bool } \quad P \Downarrow_{r} \text { false }}{\operatorname{NoSat}\left[R_{1}, R_{2}\right](P, r)}$

NoSat-Left
$\frac{\operatorname{NoSat}\left[R_{1}, R_{2}\right](P, r)}{\operatorname{NoSat}\left[R_{1}, R_{2}\right](P \wedge Q, r)}$

NoSat-Right
$\frac{\operatorname{NoSat}\left[R_{1}, R_{2}\right](Q, r)}{\operatorname{NoSat}\left[R_{1}, R_{2}\right](P \wedge Q, r)}$

We show that if a predicate $P$ satisfies the default value constraint, then the $\operatorname{NoSat}\left[R_{1}, R_{2}\right](P, r)$ is equivalent to showing that there is no value $s: R_{2}$ such that $r \otimes s$ satisfies the predicate.

Lemma 95. Assuming $\boldsymbol{D} \boldsymbol{V}^{\dagger}\left[R_{1}, R_{2}\right](P, v)$ and $r: R_{1}$. Iff. $\operatorname{NoSat}\left[R_{1}, R_{2}\right](P, r)$ then $\forall s: R_{2} . r \otimes s \Downarrow_{P}$ false.

Proof. For the first half of this proof we need to show that $\forall s: R_{2} . r \otimes s \Downarrow_{P}$ false if $\operatorname{NoSat}\left[R_{1}, R_{2}\right](P, r)$. We perform induction on the $\operatorname{NoSat}\left[R_{1}, R_{2}\right](P, r)$ property. If NoSAt-1 applies, then we know that $R_{1} \vdash P$ : bool and $P \Downarrow_{r}$ false. Using Lemma 44 it follows that $\forall s: R_{2} . r \otimes s \Downarrow_{P}$ false. The other two cases follow trivially from induction and the fact that false $\wedge v=v \wedge$ false $=$ false.

In the other direction we perform induction on $\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](P, v)$.

- DVI ${ }^{\dagger}$-1: We know that $R_{1} \vdash P$ : bool and hence NoSat-1 follows trivially from Lemma 44
- DVI ${ }^{\dagger}-2$ : We know that $R_{2} \vdash P$ : bool and $P \Downarrow_{v}$ true. Using Lemma 44 we can derive $P \Downarrow_{r \otimes v}$ true which contradicts our assumption.
- DVI ${ }^{\dagger}$-And: We know that the predicate is a conjunction $P=P^{\prime} \wedge Q$. We first argue that either $\forall s: R_{2} . P^{\prime} \Downarrow_{r \otimes s}$ false or $\forall s: R_{2} . Q \Downarrow_{r \otimes s}$ false must hold. If neither of these properties applied, then the default value property would require that both $P^{\prime} \Downarrow_{r \otimes v}$ true and $Q \Downarrow_{r \otimes v}$ true which would imply that $P \Downarrow_{r \otimes v}$ true holds, contradicting our assumption. In the case of $\forall s: R_{2} . P^{\prime} \Downarrow_{r \otimes s}$ false the induction property allows us to show $\operatorname{NoSat}\left[R_{1}, R_{2}\right]\left(P^{\prime}, r\right)$, and hence NoSAtLeft applies. In the other case, we can use the induction hypothesis to show $\operatorname{NoSat}\left[R_{1}, R_{2}\right](Q, r)$, and hence $\operatorname{NoSat}\left[R_{1}, R_{2}\right](Q, r)$ can be derived.

If the predicate satisfies our inductive property, then the rewritten predicate does not contain that record $r$.

Lemma 96. If $\operatorname{NoSat}\left[R_{1}, R_{2}\right](P, r)$, then $\llbracket P \rrbracket_{R_{1}, R_{2}} \Downarrow_{r}$ false.

Proof. Perform induction on NoSat $\left[R_{1}, R_{2}\right](P, r)$ :

- NoSAt-1: We know that $R_{1} \vdash P$ : bool, and so $\llbracket P \rrbracket_{R_{1}, R_{2}}=P$. We also know that $P \Downarrow_{r}$ false and therefore $P \Downarrow_{r}$ false.
- NoSAt-Left: In this case $P=P^{\prime} \wedge Q$ and $\llbracket P \rrbracket_{R_{1}, R_{2}}=\llbracket P^{\prime} \rrbracket_{R_{1}, R_{2}} \wedge \llbracket Q \rrbracket_{R_{1}, R_{2}}$. From the condition $\operatorname{NoSat}\left[R_{1}, R_{2}\right]\left(P^{\prime}, r\right)$ we can derive $\llbracket P^{\prime} \rrbracket R_{R_{1}, R_{2}} \Downarrow_{r}$ false using the induction hypothesis, and hence $\llbracket P \rrbracket_{R_{1}, R_{2}} \Downarrow_{r}$ false.
- NoSAt-Right: In this case $P=P^{\prime} \wedge Q$ and $\llbracket P \rrbracket_{R_{1}, R_{2}}=\llbracket P^{\prime} \rrbracket_{R_{1}, R_{2}} \wedge \llbracket Q \rrbracket_{R_{1}, R_{2}}$. From the condition $\operatorname{NoSat}\left[R_{1}, R_{2}\right](Q, r)$ we can derive $\llbracket Q \rrbracket_{R_{1}, R_{2}} \Downarrow_{r}$ false using the induction hypothesis, and hence $\llbracket P \rrbracket_{R_{1}, R_{2}} \Downarrow_{r}$ false.

Lemma 97. Iff. $\forall r: R . P \Downarrow_{r}$ false then $\operatorname{set}(P, R)=\varnothing$.

Proof. Follows trivially from the definition of set and sat.
Lemma 98. Assuming $R=R_{1} \oplus R_{2}$ and $v: R_{2}$. If $\boldsymbol{D} \boldsymbol{V}^{\dagger}\left[R_{1}, R_{2}\right](P, v)$ and $\operatorname{set}(P, R)=\varnothing$ then $\operatorname{set}\left(\llbracket P \rrbracket_{R_{1}, R_{2}}, R_{1}\right)=\varnothing$.

We can then show that if a predicate is empty, the rewritten predicate must also be empty. This means that our typing rule is correctly behaved even if the predicate is empty (which would not be permitted with the Bohannon et al. [12] rule).

Proof. From the conditions $\mathbf{D V}^{\dagger}\left[R_{1}, R_{2}\right](P, v)$ and $\operatorname{set}(P, R)=\varnothing$ we can derive $\forall r$ : $R$. $P \Downarrow_{r}$ false using Lemma 97. This allows us to derive $\forall r: R_{1}$. NoSat $\left[R_{1}, R_{2}\right](P, r)$ using Lemma 95 . We can then show that $\forall r: R_{1} \cdot \llbracket P \rrbracket_{R_{1}, R_{2}} \Downarrow_{r}$ false using Lemma 96 From Lemma 97 it follows that $\operatorname{set}\left(P, R_{1}\right)=\varnothing$.

## Appendix E

## Case Study: Curated Scientific Databases

This case study was authored by Simon Fowler.

In this section, we illustrate the use of relational lenses in the setting of a larger Links application: part of the curation interface for a scientific database. Scientific databases collect information about a particular topic, and are curated by subject matter experts who manually enter and update entries.

The IUPHAR/BPS Guide to Pharmacology (GtoPdb) [71] is a curated scientific database which collects information on pharmacological targets, such as receptors and enzymes, and ligands such as pharmaceuticals which act upon targets. GtoPdb consists of a PostgreSQL database, a Java/JSP web application frontend to the database, and a Java GUI application used for data curation.

In parallel work [39], a workalike frontend application has been developed in Links, using the Links LINQ functionality. In this section, we demonstrate how we are beginning to use relational lenses for the curation interface, and show how relational lenses are useful in tandem with the Model-View-Update (MVU) paradigm pioneered by the Elm programming language [1].

## E. 1 Disease Curation Interface

One section of GtoPdb collects information on diseases, such as the disease name, description, crossreferences to other databases, and relevant drugs and targets. In this section, we describe a curation interface for diseases, where all interaction with the


Figure E.1: Java disease curation interface


Figure E.2: Links reimplementation of the curation interface for diseases
database occurs using relational lenses.
Figure E. 1 shows the official Java curation interface. The main data entries edited using the curation interface are the name and description of the disease; the crossreferences for the disease which refer to external databases; and the synonyms for a disease. As an example, a synonym for "allergic rhinitis" is "hayfever". Note that this curation interface does not edit ligand or target information; curation of ligand-to-disease and target-to-disease links are handled by the ligand and target curation interfaces respectively.

## E. 2 Links Reimplementation

Figure E. 2 shows the curation interface as a Links web application. In the original implementation of Links [26], requests invoked Links as a CGI script. Modern Links web applications execute as follows:

1. A Links application is executed, which registers URLs against page generation functions, and starts the webserver
2. A request is made to a registered URL, and the server runs the corresponding page generation function
3. The page generation function may spawn server processes, make database queries, and register processes to run on the client, before returning HTML to the client
4. The client application spawns any client processes, and renders the HTML
5. Client processes can communicate with server processes over a WebSocket connection.

## E.2.1 Architecture

The disease curation interface consists of a persistent server process, and a client process which is spawned by the Links MVU library.

Upon page creation, the application creates lenses to the underlying tables: the lenses retrieve data from, and propagate changes to, the database. Since lenses only exist on the server and cannot be serialised to the client, we spawn a process which awaits a message from the client with the updated data.

## E.2.2 Tables and Lenses.

We begin by defining the records we need, and handles to the underlying database and its tables.

First, we define a database handle, db, to the gtopdb database.

```
var db = database "gtopdb";
```

Next, we define type aliases for the types of records in each table. The disease curation interface uses tables describing four entity types: disease data (DiseaseData), metadata about external databases (ExternalDatabase), links from diseases to external databases (DatabaseLink), and disease synonyms (Synonym). (Note that "prefix" appears in quotes as prefix is a Links keyword).

```
typename DiseaseData =
    (disease_id: Int, name: String,
    description: String, type: String);
typename ExternalDatabase =
    (database_id: Int, name: String, url: String,
    specialist: Bool, "prefix": String);
typename DatabaseLink =
    (disease_id: Int, database_id: Int, placeholder: String);
typename Synonym = (disease_id: Int, synonym: String);
```

We will need to join the ExternalDatabase and DatabaseLink tables in order to render the database name of each external database link. It is therefore useful to define a type synonym for the record type resulting from the join:

```
typename JoinedDatabaseLink =
    (disease_id: Int, database_id: Int, placeholder: String,
    name: String, url: String,
    specialist: Bool, "prefix": String);
```

Next, we can define handles to each database table. The with clause specifies a record type denoting the column name and type of each attribute in the table, and the tablekeys clause specifies the primary keys (i.e., sets of attributes which uniquely identify a row in the database) for each table. We show only the definition of diseaseTable; the definitions for databaseTable, dbLinkTable, and synonymTable are similar.

```
var diseaseTable =
    table "disease" with DiseaseData
    tablekeys [["disease_id"]] from db;
```

The ID of the disease to edit (diseaseID) is provided as a GET parameter to the page, and thus we need a dynamic predicate as not all information is known statically. With the description of the entities and tables defined, we can describe the relational lenses over the tables. We work in a function scope where diseaseID has been extracted from the GET parameters.

```
fun diseaseFilter(x) { x.disease_id == diseaseID }
# Disease lenses
var diseasesLens = lens diseaseTable default;
var diseasesLens =
    check (select from diseasesLens by diseaseFilter);(*\vspace{0.5em}*)
# Database link lenses
var dbLens = lens databaseTable
    with { database_id -> name url specialist "prefix" };
var dbLinksLens = lens dbLinkTable default;
var dbLinksLens =
    check (select from dbLinksLens by diseaseFilter);
var dbLinksJoinLens = check (
    join dbLinksLens with dbLens
        on database_id delete_left);(*\vspace{0.5em}*)
# Synonym lenses
var synonymsLens = lens synonymTable default;
var synonymsLens =
    check (select from synonymsLens by diseaseFilter);
```

We create a lens over a table using the lens keyword, writing default when we do not need to specify functional dependencies. The dbLens lens specifies a functional dependency from database_id to each of the other columns, as knowledge of this dependency is required when constructing a join lens.

We need not filter the databaseTable table since we wish to display all external databases. The diseaseLens, dbLinksLens, and synonymsLens lenses make use of the select lens combinator, allowing us to consider only the records relevant to the given diseaseID. Note that each entity has a disease_id field: as a result, we can make use of Links' row typing system [64] to define a single predicate, diseaseFilter, for each select lens using row polymorphism.

The dbLinksJoinLens lens joins the external database links with the data about each external database by using the join lens combinator, stating that if a record is deleted from the view, then it should be deleted from the dbLinkTable rather than the dbLens table. Joining these two tables is only possible because database_id uniquely determines each column of the databaseTable table; as the lens uses a dynamic predicate, this property is checked at runtime.

## E.2.3 Model

In implementing the case study, we make use of the Model-View-Update (MVU) paradigm, pioneered by the Elm programming language [1]. MVU is similar to the Model-ViewController design pattern in that it splits the state of the system from the rendering logic. In contrast to MVC, MVU relies on explicit message passing to update the model. The key interplay between MVU and relational lenses is that MVU allows the model to be directly modified in memory, and relational lenses allow the changes in the model
to be directly propagated to the database without writing any marshalling or query construction code.

```
typename DiseaseInfo =
    (diseaseData: DiseaseData, databases: [ExternalDatabase],
    dbLinks: [JoinedDatabaseLink], synonyms: [Synonym]);
typename Model =
    Maybe(
        (diseaseInfo: DiseaseInfo, selectedDatabaseID: Int,
        accessionID: String, newSynonym:String,
        submitDisease: (DiseaseInfo) {}~> ()));
```

The model (Model) contains all definitions retrieved from the database (DiseaseInfo), as well as the current value of the various form components for adding database links (selectedDatabaseID and accessionID) and synonyms (newSynonym). Finally, the model contains a function submitDisease which commits the information to the database. Note that the \{\}~> function arrow denotes a function which cannot be run on the database, and does not perform any effects. The Model type is wrapped in a Maybe constructor to handle the case where the application tries to curate a nonexistent disease.

## E.2.3.1 Initial model.

To construct the initial model, we fetch the data from each lens using the get primitive.
We include type annotations for clarity, but they are not required.

```
var (diseases: [DiseaseData]) = get diseasesLens;
var (dbs: [ExternalDatabase]) = get dbLens;
var (dbLinks: [JoinedDatabaseLink]) = get dbLinksJoinLens;
var (synonyms: [Synonym]) = get synonymsLens;
```

Next, we spawn a server process which awaits the submission of an updated DiseaseInfo record. The Submit message contains the updated record along with a client process ID notifyPid which is notified when the query is complete.

The submitDisease function takes an updated DiseaseInfo process ID and sends a Submit message to the server. The spawnWait keyword spawns a process, waits for it to complete, and returns the retrieved value. In our case, we use spawnWait to only navigate away from the page once the query has completed.

```
var pid = spawn {
    receive {
        case Submit(diseaseInfo, notifyPid) ->
            put diseasesLens with [diseaseInfo.diseaseData];
            put dbLinksJoinLens with diseaseInfo.dbLinks;
            put synonymsLens with diseaseInfo.synonyms;
            notifyPid ! Done
    }
};
sig submitDisease : (DiseaseInfo) {}~> ()
```

```
fun submitDisease(diseaseInfo) {
    spawnWait {
        pid ! Submit(diseaseInfo, self());
        receive { case Done -> () }
    };
    redirect("/editDiseases")
}
```

Given the above, we can construct the initial model. Recall that the result of get diseasesLens is a list of DiseaseInfo records. As disease_id is the primary key for the disease table, we know that the result set must be either empty or a singleton list. Finally, we can initialise the model with the data retrieved from the database along with the submitDisease function and default values for the form elements.

```
var (initialModel: Model) = {
    switch(diseases) {
        case [] -> Nothing
        case d :: _ ->
            var diseaseInfo =
                (diseaseData = d, databases = dbs,
                dbLinks = dbLinks, synonyms = synonyms);
            Just((diseaseInfo = diseaseInfo,
                    accessionID = "", newSynonym = "",
                        selectedDatabaseID = hd(dbs).database_id,
                            submitDisease = submitDisease))
    }
};
```

The model is rendered to the page using a view function which takes a model and produces some HTML to display. Interaction with the page produces messages which cause changes to the model. Finally, submission of the form causes the submitDisease function to be executed, which in turn sends a Submit message to the server to propagate the changes to the database using the lenses.

## E. 3 Discussion

In this section, we have described part of the curation interface for a scientific database. Our application is a tierless web application with the client written using the Model-View-Update architecture.

Relational lenses allow seamless integration between all three layers of the application. Lenses with dynamic predicates allow us to retrieve the relevant data from the database; the data is used as part of a model which is changed directly as a result of interaction with the web page; and the updated data entries are committed directly to the database. At no point does a user need to write a query: every interaction with the database uses only lens primitives.

The primary limitation of the implementation at present is that it does not currently support auto-incrementing primary keys, which are commonly used in relational databases.


[^0]:    ${ }^{1}$ This is slightly simpler than, but equivalent to, the definition given by Bohannon et al. [12]. Proof that the two definitions are equivalent is shown in Appendix A. 3 .

[^1]:    ${ }^{1}$ If predicate evaluation supports short-circuit evaluation, then some predicates with unknowns would be supported if they still evaluate to true.

[^2]:    ${ }^{1}$ This example does not satisfy functional dependency tree form. Even if it instead only had the functional dependencies user -> review, the same problem would occur.

[^3]:    ${ }^{1}$ http://hackage.haskell.org/package/base-4.12.0.0/docs/GHC-TypeLits.html

[^4]:    2 https://hackage.haskell.org/package/base-4.12.0.0/docs/GHC-TypeLits.html\#v:symbolVal

[^5]:    3 https://hackage.haskell.org/package/row-types
    4 https://hackage.haskell.org/package/CTRex

[^6]:    $\sqrt[5]{\text { https://hackage.haskell.org/package/postgresql-simple }}$

