Language-Integrated Updatable Views

Rudi Horn
University of Edinburgh
United Kingdom
r.horn@ed.ac.uk

Simon Fowler
University of Edinburgh
United Kingdom
simon.fowler@ed.ac.uk

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

ABSTRACT

Relational lenses are a modern approach to the *view update* problem in relational databases. As introduced by Bohannon et al. [5], relational lenses allow the definition of updatable views by the composition of lenses performing individual transformations. Horn et al. [20] provided the first implementation of *incremental relational lenses*, which demonstrated that relational lenses can be implemented efficiently by propagating *changes* to the database rather than replacing the entire database state.

However, neither approach proposes a concrete language design; consequently, it is unclear how to integrate lenses into a general-purpose programming language, or how to check that lenses satisfy the well-formedness conditions needed for predictable behaviour. In this paper, we propose the first full account of relational lenses in a functional programming language, by extending the Links web programming language. We provide support for higher-order predicates, and provide the first account of typechecking relational lenses which is amenable to implementation. We prove the soundness of our typing rules, and illustrate our approach by implementing a curation interface for a scientific database application.

ACM Reference Format:
Rudi Horn, Simon Fowler, and James Cheney. 2020. Language-Integrated Updatable Views. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

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

Programs interface with relational databases using the *Structured Query Language* (SQL). To query the database, the host application needs to generate an SQL query from user input, issue it to the database server, and then process the result in a way that aligns with the result of the query.

As an example, we consider a music database, originally proposed by Bohannon et al. [5] and shown in Figure 1. There are two tables: the `albums` table, which details the quantities of albums available, and the `tracks` table, which details the track name, year of release, rating, and the album on which the track is contained. Our application could generate a query by using string concatenation

`albums`

album	quantity
Disintegration	6
Show	3
Galore	1
Paris	4
Wish	5

`tracks`

track	year	rating	album
Lullaby	1989	3	Galore
Lullaby	1989	3	Show
Lovesong	1989	5	Galore
Lovesong	1989	5	Paris
Trust	1992	4	Wish

Figure 1: Music Database

and then assume the result will be in a known format containing records of track names of type **string** and years of type **int**.

However, such an approach leaves many possible sources of error, most of which are related to a lack of cross-checking of the different stages of execution. The application could have bugs in query generation, which might result in incorrect queries or even security flaws. Furthermore, a generated query may not produce a result of the type that the application expects, resulting in a runtime error. The user experience of the programmer is also poor, as tooling provides little help and the programmer must write code in two different languages, while being mindful not to introduce any bugs in the application. We refer to this as an *impedance mismatch* between the host programming language and SQL [9].

Existing work on *language integrated query* (LINQ) allows queries to be expressed in the host language [8, 30]. Rather than generating an SQL query using string manipulation, the query is written in the same syntax as the host programming language. The user need not worry about how the query is generated, and the code that performs the database query is automatically type-checked at compile time.

As an example of LINQ, consider the following function, written in the Links [8] programming language, which queries the `albums` table and returns all albums with a given album name:

```
fun getAlbumsByName(albumName) {
  for (a <-- albums)
    where (a.album == albumName)
    [a]
}
```

The corresponding SQL for `getAlbumsByName("Galore")` would be:

```
SELECT * FROM albums AS a WHERE a.album == "Galore"
```

IFL'19, September 2019, Singapore

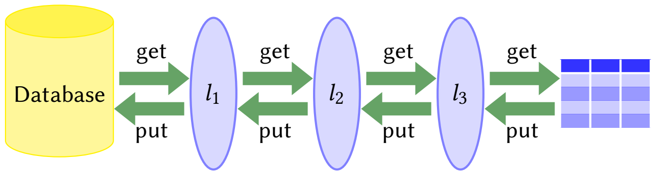


Figure 2: Relational Lenses

LINQ approaches are convenient for querying databases, but still take a relatively fine-grained approach to data manipulation (updates). The programmer is required to explicitly determine which changes were made at the application level. All modifications made by the user must then be translated into equivalent insertions, updates and deletions for each table. In contrast, a typical user workflow consists of fetching a subset of the database, called a *view*, making changes to this view, and then propagating the changes to the database. Defining views that can be updated directly is known as the *view-update problem*, a long-standing area of study in the field of databases [4].

Relational Lenses. A recent approach to the view update problem is to define views using composable *relational lenses* [5]. Lenses are a form of *bidirectional transformation* [14]. With relational lenses, instead of defining the view using a general SQL query, the programmer defines the view by combining individual lenses, which are known to behave in a correct manner. Bohannon et al. [5] define lenses for relational algebra operations, in particular, projections, selections and joins. Figure 2 shows the composable nature of relational lenses.

A relational lens can be considered a form of *asymmetric* lens, in which we have a forward (*get*) direction to fetch the data, and a reverse (*put*) direction to make updates [16]. A bidirectional transformation is *well-behaved* if it satisfies round-tripping guarantees:

$$\textsc{GetPut}\quad \text{put}\ s\ (\text{get}\ s) = s \qquad \textsc{PutGet}\quad \text{get}\ (\text{put}\ s\ v) = v$$

Relational lenses are equipped with typing rules which ensure operations on lenses are well-behaved. The type system for relational lenses tracks the attribute types of the defined view as well as constraints, including *predicates* and *functional dependencies*, which are not easily expressible in an ML-like type system.

From theory to practice. The theory of relational lenses was developed over a decade ago by Bohannon et al. [5], but until recently there has been little work on practical implementations. Horn et al. [20] recently presented the first implementation using an incremental semantics. However Horn et al. [20] focus on performance rather than language integration, leaving two issues unresolved:

- How to integrate relational lenses, which are defined as a sequential composition of primitives, into a functional language, where lenses are composed using lens subexpressions.
- How to define and verify the correctness of a concrete *selection predicate* syntax for relational lenses.

Predicates. Some of the relational lens constructors, such as the *select* lens, require user supplied functions for filtering rows. Such functions, called *predicates*, determine whether or not an individual record should be included. Predicates are a function of type $R \to$ **bool** where R is the type of the input record and a return value of **true** indicates that the predicate holds.

Bohannon et al. [5] treat predicates as abstract (finite or infinite) sets, without giving a computational syntax. Sets allow predicates to be defined in an abstract form while still being amenable to mathematical reasoning, but such an approach does not scale to a practical implementation. In practice the user should define a predicate as a function from a record (in this case containing `album` and `year` fields) to a Boolean value:

```
fun(x) { x.album = "Galore" && x.year == 1989 }
```

Some of the lens typing rules require static checks on predicates. The above predicate contains only *static* information, and is thus a *closed function* which can be checked at compile-time. We call such predicates *static* predicates. Alas, such checks become problematic when the programmer would like to define a function which depends on information only available at runtime, such as a parameter in a web request. For example, consider the following function which adapts the `getAlbumsByName` function to use relational lenses.

```
fun getAlbumsByNameL(albumName) {
  var albumLens = lens albums where album -> quantity;
  var selectLens = select from albumsLens where
    fun(a) { a.album == albumName };
  get selectLens;
}
```

The `getAlbumsByNameL` function begins by defining `albumLens` as a lens over the `albums` table. A *functional dependency* $\vec{\ell} \to \vec{\ell'}$ states that the columns in $\vec{\ell'}$ are *uniquely determined* by $\vec{\ell}$; here, the `album -> quantity` clause states that the `quantity` attribute is uniquely determined by the `album` attribute.

As `albumName` is supplied as a parameter to the `getAlbumsByNameL` function, the anonymous predicate supplied to **`select`** can only be completely known at runtime. We call such predicates *dynamic* predicates. Dynamic predicates are not closed, which means that variables in the closure of a dynamic predicate may not be available until runtime, and may themselves refer to functions. While it is possible to statically know the *type* of the function, and thus rule out a class of errors, relational lenses require finer-grained checks which require a more in-depth analysis of the predicate. As an example, a select lens is only well-formed if the predicate does not rely on the output of a functional dependency.

If we required the function to be fully known at compile time, a programmer could not define predicates that depend on user input. Thus, there is a tradeoff between static correctness and programming flexibility. In our design, we can perform checks on lenses using static predicates at compile-time, and we can also support dynamic predicates by performing the same checks at runtime.

Another obstacle is the handling of functional dependencies, which are an important part of the type system for relational lenses. Functional dependencies are constraints that apply to the data, and specify which fields in a table uniquely determine other fields.

The typing rules given by Bohannon et al. [5] are important for showing soundness of relational lenses: without ensuring all the requirements are met, it is not possible to ensure the lenses are well-behaved. We take the existing work by Bohannon et al. [5] and concretise and adapt the design to allow the rules to be implemented in practice.

1.1 Contributions

The primary technical contribution of this paper is the first full design and implementation of relational lenses in a typed functional programming language, namely Links [8]. This paper makes three concrete contributions:

(1) A design and implementation of *predicates* for relational lenses, based on previous approaches to language-integrated query. We define a language of predicates, and show how terms can be normalised to a fragment both amenable to typechecking of relational lenses, and translation to SQL.
(2) An implementation of the typing rules for relational lenses, adapted to the setting of a functional programming language (§3). We prove (§3.4) that our compositional typing rules are sound with respect to the original rules proposed by Bohannon et al. [5]. Static predicates can be fully checked at compile time, whereas the same checks can be performed on dynamic predicates at runtime.
(3) A curation interface for a real-world scientific database implemented as a cross-tier web application, tying together relational lenses with the Model-View-Update architecture for frontend web development (§4).

We have packaged our implementation and example application as an artifact [18]. Proofs of the technical results can be found in the extended version of the paper [19].

The remainder of the paper proceeds as follows: §2 describes the design and implementation of predicates; §3 describes the implementation of static typechecking for relational lenses; §4 describes the case study; §5 describes related work; and §6 concludes.

2 PREDICATES

In their original proposal for relational lenses, Bohannon et al. [5] define predicates using abstract sets. Although theoretically convenient, such a representation is not suited to implementation in a programming language. Our first task in implementing relational lenses, therefore, is to define a concrete syntax for predicates.

As we are working in the setting of a functional programming language, it is natural to treat predicates as functions from records to Boolean values. As an example, recall our earlier example of the `select` lens, which selects albums with a given name:

```
select from albumsLens where fun(a) { a.album == albumName }
```

Here, the predicate function is `fun(a) {a.album == albumName}`. Intuitively, this predicate includes a record `a` in the set of results if its `album` field matches `albumName`.

In our approach, predicates are a well-behaved subset of Links functions which take a parameter of the type of row on which the lens operates. We define a simply-typed λ-calculus for predicates, and apply the normalisation approach advocated by Cooper [7] to derive a form which is both amenable to SQL translation, and can be used when typechecking lens construction.

2.1 Static and Dynamic Predicates

Ensuring relational lenses are well-typed requires some conditions that require static knowledge of predicates. 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; we describe the conditions more in detail in Section 3.

Syntax

$$
\begin{array}{llcl}
\text{Types} & A, B, C & ::= & A \to B \mid (\overrightarrow{\ell : A}) \mid D \\
\text{Base types} & D & ::= & \textbf{bool} \mid \textbf{int} \mid \textbf{string} \\
\text{Base record types} & R & ::= & (\overrightarrow{\ell : D}) \\
 & & & \\
\text{Labels} & \ell & & \\
\text{Terms} & L, M, N & ::= & x \mid c \mid \lambda x.\, M \mid M\, N \\
 & & \mid & (\overrightarrow{\ell = M}) \mid M.\ell \\
 & & \mid & \textbf{if } L \textbf{ then } M \textbf{ else } N \\
 & & \mid & \odot\{\overrightarrow{M}\}
\end{array}
$$

Typing rules

$$
\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{T-Var}
\qquad
\frac{c \text{ of type } A}{\Gamma \vdash c : A}\ \textsc{T-Const}
\qquad
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \textsc{T-Abs}
$$

$$
\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}\ \textsc{T-App}
\qquad
\frac{\begin{array}{c}(\Gamma \vdash M_i : A_i)_i \\ \text{for each } M_i : A_i \in \overrightarrow{M : A}\end{array}}{\Gamma \vdash (\overrightarrow{\ell = M}) : (\overrightarrow{\ell : A})}\ \textsc{T-Record}
$$

$$
\frac{\Gamma \vdash M : (\ell_i : A_i)_{i \in I} \qquad j \in I}{\Gamma \vdash M.\ell_j : A_j}\ \textsc{T-Project}
\qquad
\frac{\begin{array}{c}\Gamma \vdash L : \textbf{bool} \\ \Gamma \vdash M : A \qquad \Gamma \vdash N : A\end{array}}{\Gamma \vdash \textbf{if } L \textbf{ then } M \textbf{ else } N : A}\ \textsc{T-If}
$$

$$
\frac{\odot : D_1 \times \ldots \times D_n \to D \qquad (\Gamma \vdash M_i : D_i)_{i \in 1..n}}{\Gamma \vdash \odot\{\overrightarrow{M}\} : D}\ \textsc{T-Op}
$$

Figure 3: Syntax and typing rules for predicate language

Our approach distinguishes two types of predicates: *static* predicates, which rely on only static information; and *dynamic* predicates, which can refer to arbitrary free variables. Referring to our previous example, `fun(a) { a.album == albumName }` is a dynamic predicate, as `albumName` is a free variable, whereas `fun(a) { a.album == "Paris"}` is a static predicate.

We can check the construction of lenses with static predicates entirely statically, whereas lenses with dynamic predicates require the same checks to be performed dynamically. Our formal results are based on static predicates, however the same results apply for dynamic predicates (which can be treated as closed at runtime).

2.2 Predicate Language

Syntax. Figure 3 shows the syntax of the predicate language. Types, ranged over by A, B, C, include function types $A \to B$; record types $(\overrightarrow{\ell : A})$ mapping labels ℓ to values of type A; and base types D, ranging over the types of Boolean values, strings, and integers. It is convenient to let R range over records whose fields are of base type. The unit type $()$ is definable as a record with no fields.

Normal forms

$$\begin{array}{rcl} O & ::= & x \mid c \mid \lambda x.O \mid (\overrightarrow{\ell = O}) \mid x.\ell \\ & \mid & \textbf{if } O_1 \textbf{ then } O_2 \textbf{ else } O_3 \mid \odot\{\overrightarrow{O}\} \\ P, Q & ::= & \textbf{if } P_1 \textbf{ then } P_2 \textbf{ else } P_3 \mid \odot\{\overrightarrow{P}\} \mid x.\ell \mid c \end{array}$$

Normalisation $\boxed{M : A \rightsquigarrow N}$

$$\begin{array}{rcl} (\lambda x.N)\, M : A & \rightsquigarrow & N[M/x] \\ (\overrightarrow{\ell = M}).\ell : A & \rightsquigarrow & M_\ell \\ \textbf{if true then } L \textbf{ else } M : A & \rightsquigarrow & L \\ \textbf{if false then } L \textbf{ else } M : A & \rightsquigarrow & M \\ (\textbf{if } L \textbf{ then } M \textbf{ else } M')\, N : A & \rightsquigarrow & \textbf{if } L \textbf{ then } M\, N \textbf{ else } M'\, N \\ \textbf{if } L \textbf{ then } M \textbf{ else } M' : (\overrightarrow{\ell : A}) & \rightsquigarrow & (\overrightarrow{\ell = N}) \\ & & \text{with } N_\ell = \\ & & \quad \textbf{if } L \textbf{ then } M.\ell \textbf{ else } M'.\ell \\ & & \text{for each } \ell \in \overrightarrow{\ell} \end{array}$$

Evaluation $\boxed{M \Downarrow V}$

$$\text{Values} \quad V \quad ::= \quad c \mid \lambda x.M \mid (\overrightarrow{\ell = V})$$

$$\frac{}{V \Downarrow V} \qquad \frac{L \Downarrow \lambda x.N \quad M \Downarrow V \quad N[V/x] \Downarrow W}{L\, M \Downarrow W} \qquad \frac{(M_i \Downarrow V_i)_i}{(\overrightarrow{\ell = M}) \Downarrow (\overrightarrow{\ell = V})}$$

$$\frac{M \Downarrow ((\ell_i = V_i)_{i \in I}) \qquad j \in I}{M.\ell_j \Downarrow V_j} \qquad \frac{L \Downarrow \textbf{true} \qquad M \Downarrow V}{\textbf{if } L \textbf{ then } M \textbf{ else } N \Downarrow V}$$

$$\frac{L \Downarrow \textbf{false} \qquad N \Downarrow V}{\textbf{if } L \textbf{ then } M \textbf{ else } N \Downarrow V} \qquad \frac{(M_i \Downarrow V_i)_i}{\odot\{\overrightarrow{M}\} \Downarrow \hat{\odot}\{\overrightarrow{V}\}}$$

Figure 4: Normalisation and Evaluation

Terms, ranged over by L, M, N, are those of the simply-typed λ-calculus extended with base types, records, conditional statements, and n-ary operators on base types $\odot\{\overrightarrow{M}\}$. We assume that the set of available operators all have an SQL equivalent and assume the existence of at least the comparison operators $<, >, ==$ and Boolean negation, conjunction, and disjunction. We sometimes find it convenient to use infix notation for binary operators.

Typing. Most typing rules are standard for the simply-typed λ-calculus extended with records. The only non-standard rule is T-Op, which states that the arguments to an operator must be of base type and match the type of the operator.

Normalisation. Given a functional language for predicates, we wish to show that predicates can be normalised to a fragment easily translatable to SQL and usable when typechecking lenses. Figure 4 introduces normal forms O which include variables, constants, λ-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 P, are a restriction of terms in normal forms. Terms in predicate normal form have a straightforward SQL equivalent, and can be used when typechecking lenses.

Normalisation rules $M \rightsquigarrow N$ are a subset of the rules proposed by Cooper [7]: the first four rules are standard β-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 1 (Strong normalisation). *If $\Gamma \vdash M : A$, then there are no infinite $\rightsquigarrow$ sequences from M.*

Proof. A special case of the result shown by Cooper [7]. □

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 2 (Normal forms). *If $x : R \vdash M : A$ and $M \rightsquigarrow^* N \not\rightsquigarrow$, then N is in normal form.*

Proof. By induction on the derivation of $x : R \vdash M : A$. □

As a corollary, by considering only terms with type **bool**, we can show that static predicates are in predicate normal form.

Corollary 3 (Predicate normal form). *If $x : R \vdash M :$ **bool** and $M \rightsquigarrow^* N \not\rightsquigarrow$, then N is in predicate normal form.*

Consequently, any static predicate written in our predicate language can be normalised to predicate normal form, allowing it to be used in typechecking of lenses and for translation into SQL. Furthermore, the normalisation procedure can be applied to any *dynamic* predicate at runtime in order to allow the same checks to be performed dynamically.

Evaluation. Figure 4 also introduces a standard big-step evaluation relation $M \Downarrow V$, which states that term M evaluates to a value V. We use the notation $\hat{\odot}\{\overrightarrow{V}\}$ to describe the denotation of operation $\odot$ applied to arguments $\overrightarrow{V}$: for example, $\hat{+}\{5, 10\} = 15$. The semantics enjoys a standard type soundness property.

Proposition 4 (Type Soundness). *If $\cdot \vdash M : A$, then there exists some V such that $M \Downarrow V$ and $\cdot \vdash V : A$.*

3 TYPECHECKING RELATIONAL LENSES

In this section, we show how naïve composition of lens combinators can give rise to ill-formed lenses, and show how such ill-formed lenses can be ruled out using static and dynamic checks. We adapt the rules proposed by Bohannon et al. [5] to the setting of a functional programming language. We begin by discussing functional dependencies, and then look at each lens combinator in turn.

3.1 Functional Dependencies

Functional dependencies are constraints restricting combinations of records. A functional dependency $\overrightarrow{\ell} \to \overrightarrow{\ell'}$ requires that two records with the same values for $\overrightarrow{\ell}$ should have the same values for $\overrightarrow{\ell'}$. We use $\mathcal{F}$ and $\mathcal{G}$ to denote sets of functional dependencies. It is possible to derive functional dependencies from other functional dependencies. The judgement $\mathcal{F} \vDash \overrightarrow{\ell} \to \overrightarrow{\ell'}$ specifies that the functional dependency $\overrightarrow{\ell} \to \overrightarrow{\ell'}$ can be derived from the set of functional

$$\begin{array}{llcl}
\text{Table names} & S, T & & \\
\text{Types} & A, B & ::= & \cdots \mid \textbf{table of } (S, R) \mid \textbf{record set of } R \\
 & & \mid & \textbf{lens of } (\Sigma, R, \lambda x.\, P, \mathcal{F}) \\
\text{Terms} & L, M, N & ::= & \cdots \mid \textbf{table } S \textbf{ with } R \mid \textbf{lens } M \textbf{ with } \mathcal{F} \\
 & & \mid & \textbf{select}_{\lambda x.\, P} \textbf{ from } M \\
 & & \mid & \textbf{join } M \textbf{ with } N \textbf{ delete\_left} \\
 & & \mid & \textbf{drop } \ell' \textbf{ determined by } (\overrightarrow{\ell}, V) \textbf{ from } M \\
 & & \mid & \textbf{get } M \mid \textbf{put } M \textbf{ with } N
\end{array}$$

Figure 5: Syntax of types and terms for tables and lenses

dependencies $\mathcal{F}$ following *Armstrong's axioms* [2]; these (standard) derivation rules can be found in the extended version of the paper. The *output fields* of the functional dependencies $\mathcal{F}$, written outputs($\mathcal{F}$), is the set of fields constrained by $\mathcal{F}$ and is defined as:

Definition 1 (Output fields).
$\text{outputs}(\mathcal{F}) = \{\ell \in \overrightarrow{\ell} \mid \exists \overrightarrow{\ell'} \in \overrightarrow{\ell}.\, \ell \notin \overrightarrow{\ell'} \textit{ and } \mathcal{F} \models \overrightarrow{\ell'} \to \ell\}$.

Bohannon et al. [5] impose a special restriction on functional dependencies called *tree form*. Tree form requires that functional dependencies form a forest, meaning that column names can be partitioned into pairwise-disjoint sets forming a directed acyclic graph with at most one incoming edge per node. As an example, $\{A \to B, A \to C, C \to D\}$ is in tree form. It is straightforward to check whether a set of functional dependencies is in tree form using a standard graph reachability algorithm.

Sets of functional dependencies which are semantically equivalent to a set of functional dependencies in tree form are also considered to be in tree form. As an example, $\{A \to BC, C \to D\}$ is not literally in tree form but is semantically equivalent to the previous example, so can thus considered to be in tree form.

3.2 Lens Types

Figure 5 shows the additional types and terms for tables and lens constructs. We let S, T range over table names. Type **table of** (S, R) is the type of a table with table name S containing records of type R. The *record set type* **record set of** R describes a set of records of type R. The type of lenses, **lens of** $(\Sigma, R, \lambda x.\, P, \mathcal{F})$, consists of four components: the set of underlying tables Σ; the base record type R; a *restriction predicate* $\lambda x.\, P$; and a set of functional dependencies $\mathcal{F}$. 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.

In the remainder of the section, we describe each lens combinator and its typing rule in turn.

3.3 Rules

We now introduce the rules we use to typecheck relational lenses, adapted from the rules as defined by Bohannon et al. [5] to support nested composition and to make use of our concrete predicate syntax. We show a formal correspondence between our typing rules and the typing rules of Bohannon et al. [5] in §3.4. We first introduce some notation.

Definition 2 (Record concatenation).

- *Given records* $r = (\ell_1 = V_1, \ldots, \ell_m = V_m)$ *and* $s = (\ell_{m+1} = V_{m+1}, \ldots, \ell_n = V_n)$ *with disjoint field names, define the* record concatenation $r \otimes s = (\ell_1 = V_1, \ldots, \ell_n = V_n)$.
- *Given record types* $R = (\ell_1 : A_1, \ldots, \ell_m : A_m)$ *and* $R' = (\ell_{m+1} : A_{m+1}, \ldots, \ell_n : A_n)$ *with disjoint field names, define the* record type concatenation $R \oplus R' = (\ell_1 : A_1, \ldots, \ell_n : A_n)$.

Tables. Links defines a primitive table expression **table** S **with** R which defines 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.

$$\textsc{T-Table}\quad \frac{}{\Gamma \vdash \textbf{table } S \textbf{ with } R : \textbf{table of } (S, R)}$$

Lens Primitives. 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 $\mathcal{F}$, written names($\mathcal{F}$), should be part of the table record type R.

$$\textsc{T-Lens}\quad \frac{\Gamma \vdash M : \textbf{table of } (S, R) \qquad \bigcup \text{names}(\mathcal{F}) \subseteq \text{dom}(R)}{\Gamma \vdash \textbf{lens } M \textbf{ with } \mathcal{F} : \textbf{lens of } (\{S\}, R, \lambda x.\, \textbf{true}, \mathcal{F})}$$

3.3.1 Select Lens.

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	year	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	year	rating	album	quantity
Lullaby	1989	3	Galore	1
Lovesong	1989	~~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	year	rating	album	quantity
Lullaby	1989	3	Galore	1
Lovesong	1989	~~5~~ 4	Galore	1
Lovesong	1989	~~5~~ 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 $\mathcal{F}$ is outputs($\mathcal{F}$). A predicate P ignores the set $\overrightarrow{\ell}$ if the result of evaluating the predicate P with respect to a row in the database is not affected by changing any fields in $\overrightarrow{\ell}$.

Definition 3 (Predicate Ignores). *We say P* ignores $\overrightarrow{\ell}$ *if there exists an R such that $\overrightarrow{\ell}$ is disjoint from* $\mathrm{dom}(R)$ *and* $x : R \vdash P : \textbf{bool}$.

The T-Select rule also needs to ensure that the resulting lens only accepts records that satisfy the given predicate $\lambda x.\,Q$ as well as any existing constraints $\lambda x.\,P$ that already apply to the underlying lens. The resulting lens's constraint predicate can thus be defined as $\lambda x. P \wedge Q$. The full select lens typing rule can be defined as:

$$\frac{\begin{array}{c}\Gamma \vdash M : \textbf{lens of } (\Sigma, R, \lambda x.\, P, \mathcal{F}) \qquad x : R \vdash Q : \textbf{bool} \\ \mathcal{F} \text{ is in tree form} \qquad P \text{ ignores outputs}(\mathcal{F})\end{array}}{\Gamma \vdash \textbf{select}_{\lambda x.\, Q} \textbf{ from } M : \textbf{lens of } (\Sigma, R, \lambda x.\, P \wedge Q, \mathcal{F})}$$

3.3.2 Join Lens. The *join* lens joins 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].

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	year	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 $\overrightarrow{\ell} \cap \overrightarrow{\ell'} \to \overrightarrow{\ell'}$, where $\overrightarrow{\ell} \cap \overrightarrow{\ell'}$ are the join columns and $\overrightarrow{\ell'}$ 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 $\overrightarrow{\ell} \cap \overrightarrow{\ell'}$ and then checking if $\overrightarrow{\ell'}$ 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. The type checking for each variant is similar, so we only discuss the delete left lens. The rule T-Join-Left requires us to also show that P ignores outputs($\mathcal{G}$) and Q ignores outputs($\mathcal{G}$). 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

$$\frac{\begin{array}{c}\Gamma \vdash M : \textbf{lens of } (\Sigma, R, \lambda x.\, P, \mathcal{F}) \qquad \Gamma \vdash N : \textbf{lens of } (\Delta, R', \lambda x.\, Q, \mathcal{G}) \\ \mathcal{G} \vDash \mathrm{dom}(R) \cap \mathrm{dom}(R') \to \mathrm{dom}(R') \\ \mathcal{F} \text{ is in tree form} \qquad \mathcal{G} \text{ is in tree form} \\ P \text{ ignores outputs}(\mathcal{F}) \qquad Q \text{ ignores outputs}(\mathcal{G}) \qquad \Sigma \cap \Delta = \emptyset\end{array}}{\Gamma \vdash \textbf{join } M \textbf{ with } N \textbf{ delete\_left} : \textbf{lens of } (\Sigma \cup \Delta, R \oplus R', \lambda x. P \wedge Q, \mathcal{F} \cup \mathcal{G})}$$

3.3.3 Drop Lens. The *drop* lens allows a more fine-grained notion of relational projection, allowing us to remove a column from a view. Note that this is not to be confused with the SQL `DROP` statement, which deletes a table. Let us assume we define the lens l_1 as a select lens with predicate `year > 1990` ∨ `rating > 4`.

track	year	rating	album
Lovesong	1989	5	Galore
Lovesong	1989	5	Paris
Trust	1992	4	Wish

We can then define the lens l_2 as l_1, but dropping column `year` determined by `track` to yield the table:

track	rating	album
Lovesong	~~5~~ 3	Galore
Lovesong	~~5~~ 3	Paris
Trust	4	Wish

What would the new predicate constraint be? It cannot reference the field `year`, 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 year 1989 and the rating is only 3.

The underlying issue is the dependency between the dropped field `year` 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 `year`. Without being able to construct such a P, a lens cannot be well-typed.

Lossless Join Decomposition. The typing rule for the drop lens requires some finer-grained checks on predicates. We begin with some preliminary definitions.

Definition 4 (Predicate satisfaction). *We say that a record r* satisfies *predicate $\lambda x.\ P$, written* $\mathrm{sat}(\lambda x.\, P, r)$*, if $P[r/x] \Downarrow$ **true**.*

Definition 5 (Record type inhabitants). *We define the* inhabitants *of a record type R, written* $\mathrm{inh}(R)$*, as:*

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

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

[1]This example does not satisfy functional dependency tree form. If it instead only had the functional dependencies `user -> review`, the same problem would occur.

Definition 6 (Predicate sets). *We define the* set representation of predicate $\lambda x.\ P$ over R, *written* $\mathsf{set}(\lambda x.\ P, R)$, *as:*

$$\{r \in \mathsf{inh}(R) \mid \mathsf{sat}(\lambda x.\ P, r)\}$$

It is often helpful to consider only a subset of fields in a record.

Definition 7 (Record restriction). *Given a record* $r = (\ell_1 = V_1, \ldots, \ell_m = V_m, \ldots, \ell_n = V_n)$, *we define the* record restriction of r to $\ell_1, \ldots, \ell_m$, *written* $r[\ell_1, \ldots, \ell_m]$, *as* $(\ell_1 = V_1, \ldots, \ell_m = V_m)$.

Let Π, Π' range over homogeneous sets of records, such as the set representation of predicates. It is also convenient to be able to consider a set where each constituent record is restricted to a given set of fields.

Definition 8 (Predicate set restriction). *We define the* restriction of set Π to $\overrightarrow{\ell}$, *written* $\Pi[\overrightarrow{\ell}]$, *as* $\{r[\overrightarrow{\ell}] \mid r \in \Pi\}$.

It is also useful to be able to consider the natural join of two sets of records.

Definition 9 (Set join). *Suppose* $R = R_1 \oplus R_2$, *and suppose* Π *contains records of type* R_1 *and* Π' *contains records of type* R_2.

We define the set join *of* Π *and* Π', *written* $\Pi \bowtie \Pi'$, *as:*

$$\{r \in \mathsf{inh}(R) \mid r[\mathsf{dom}(R_1)] \in \Pi \land r[\mathsf{dom}(R_2)] \in \Pi'\}$$

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* (LJD).

Definition 10 (Lossless join decomposition). *A* lossless join decomposition *of two record types* R_1 *and* R_2 *with respect to a predicate* P *of type* $x : R_1 \oplus R_2 \vdash P : \textbf{bool}$, *written* $\textbf{LJD}_{R_1,R_2}\ (\lambda x.\ P)$, *means that for all* $r_1, r_2 \in \mathsf{inh}(R_1)$ *and* $s_1, s_2 \in \mathsf{inh}(R_2)$, *it is the case that:*

$$\mathsf{sat}(\lambda x.\ P, r_1 \otimes s_1) \land \mathsf{sat}(\lambda x.\ P, r_2 \otimes s_2) \implies \mathsf{sat}(\lambda x.\ P, r_1 \otimes s_2)$$

Given R, R_1, R_2 such that $R = R_1 \oplus R_2$, our definition of lossless join decomposition suffices to show that $\mathsf{set}(\lambda x.\ P, R)$ can be expressed as the natural join of $\mathsf{set}(\lambda x.\ P, R)$ restricted to the fields of R_1, with $\mathsf{set}(\lambda x.\ P, R)$ restricted to the fields of R_2.

Lemma 1 (Predicate set decomposition). *Suppose* $R = R_1 \oplus R_2$ *and* $x : R \vdash P : \textbf{bool}$. *If* $\textbf{LJD}_{R_1,R_2}\ (\lambda x.\ P)$, *then* $\mathsf{set}(\lambda x.\ P, R) = \mathsf{set}(\lambda x.\ P, R)[\mathsf{dom}(R_1)] \bowtie \mathsf{set}(\lambda x.\ P, R)[\mathsf{dom}(R_2)]$.

Proof. Follows from the definitions of $\textbf{LJD}_{R_1,R_2}\ (\lambda x.\ P)$, $\cdot[\cdot]$ and $\cdot \bowtie \cdot$. □

Showing $\textbf{LJD}_{R,R'}\ (\lambda x.\ P)$ is NP-hard and could be undecidable, depending on the atomic formulae available in the predicates. Since a predicate that satisfies $\textbf{LJD}_{R,R'}\ (\lambda x.\ P)$ can be rewritten as a conjunction of predicates which depend only on either R or R', we can, however, define a sound but incomplete syntactic approximation $\textbf{LJD}^{\dagger}_{R,R'}\ (\lambda x.\ P)$.

LJD†-1
$$\frac{x : R \vdash P : \textbf{bool}}{\textbf{LJD}^{\dagger}_{R,R'}\ (\lambda x.\ P)}$$

LJD†-2
$$\frac{x : R' \vdash P : \textbf{bool}}{\textbf{LJD}^{\dagger}_{R,R'}\ (\lambda x.\ P)}$$

LJD†-And
$$\frac{\textbf{LJD}^{\dagger}_{R,R'}\ (\lambda x.\ P) \qquad \textbf{LJD}^{\dagger}_{R,R'}\ (\lambda x.\ Q)}{\textbf{LJD}^{\dagger}_{R,R'}\ (\lambda x.\ P \land Q)}$$

Lemma 2 (Soundness of LJD†). *Given a predicate* $\lambda x.\ P$ *and record types* R, R', *it follows that* $\textbf{LJD}^{\dagger}_{R,R'}\ (\lambda x.\ P)$ *implies* $\textbf{LJD}_{R,R'}\ (\lambda x.\ P)$.

Proof. By induction on the derivation of $\textbf{LJD}^{\dagger}_{R,R'}\ (\lambda x.\ P)$. □

Updates to the view will use the default value V in place of the given column. 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 value V of the dropped column does not violate the predicate. Given the set representation of a predicate $\mathsf{set}(\lambda x.\ P, R)$, we must show that $\{\ell' = V\} \in \mathsf{set}(\lambda x.\ P, R)[\ell']$.

We define a property $\textbf{DV}_{R,R'}\ (\lambda x.\ P)\ r$ and show that it is sound with respect to the set semantics.

Definition 11. *Given a predicate* $\lambda x.\ P$ *and record types* R *and* R' *such that* $\textbf{LJD}_{R,R'}\ (\lambda x.\ P)$ *and* $r \in \mathsf{inh}(R')$, *we write* $\textbf{DV}_{R,R'}\ (\lambda x.\ P)\ r$ *when* $\mathsf{set}(\lambda x.\ P, R \oplus R')$ *is not empty and there exists an* $s \in \mathsf{inh}(R)$ *such that* $\mathsf{sat}(\lambda x.\ P, r \otimes s)$.

Lemma 3. *Suppose* $R = R_1 \oplus R_2$, $r \in \mathsf{inh}(R_2)$, *and* $\textbf{LJD}_{R_1,R_2}\ (\lambda x.\ P)$. *Then* $\textbf{DV}_{R_1,R_2}\ (\lambda x.\ P)\ r$ *implies* $r \in \mathsf{set}(\lambda x.\ P, R)[\mathsf{dom}(R_2)]$.

Proof. By expansion of the definitions of $\textbf{DV}_{R_1,R_2}\ (\lambda x.\ P)\ r$ and of $\cdot \otimes \cdot$ and $\cdot \in \cdot$. □

As with the definition of $\textbf{LJD}_{R,R'}\ (\lambda x.\ P)$, determining if $\textbf{DV}_{R,R'}\ (\lambda x.\ P)\ r$ holds in the general case is a difficult problem. To simplify this problem we introduce an incomplete set of inference rules to determine $\textbf{DV}^{\dagger}_{R,R'}\ (\lambda x.\ P)\ r$, which covers the same set of predicates as the $\textbf{LJD}^{\dagger}_{R,R'}\ (\lambda x.\ P)$ rule.

DV†-1
$$\frac{x : R \vdash P : D}{\textbf{DV}^{\dagger}_{R,R'}\ (\lambda x.\ P)\ r}$$

DV†-2
$$\frac{x : R' \vdash P : D \qquad \mathsf{sat}(\lambda x.\ P, r)}{\textbf{DV}^{\dagger}_{R,R'}\ (\lambda x.\ P)\ r}$$

DV†-And
$$\frac{\textbf{DV}^{\dagger}_{R,R'}\ (\lambda x.\ P)\ r \qquad \textbf{DV}^{\dagger}_{R,R'}\ (\lambda x.\ Q)\ r}{\textbf{DV}^{\dagger}_{R,R'}\ (\lambda x.\ P \land Q)\ r}$$

Lemma 4. *Given a predicate* $\lambda x.\ P$ *such that* $\mathsf{set}(\lambda x.\ P, R \oplus R')$ *is not empty and record* r *such that* $\cdot \vdash r : R$, *it follows that* $\textbf{DV}^{\dagger}_{R,R'}\ (\lambda x.\ P)\ r$ *implies* $\textbf{DV}_{R,R'}\ (\lambda x.\ P)\ r$.

Proof. By induction on the derivation of $\textbf{DV}^{\dagger}_{R,R'}\ (\lambda x.\ P)\ r$. □

Note that the soundness proof for $\textbf{DV}^{\dagger}_{R,R'}\ (\lambda x.\ P)\ r$ requires that $\mathsf{set}(\lambda x.\ P, R \oplus R')$ is not empty. This is problematic in theory, because it requires us to show that the predicate is satisfiable. According to Bohannon et al. [5], 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 and only takes an empty view. The lens would therefore be useless, but not incorrect.

With the preliminaries in place, we can present the typing rule for the drop lens. The term **drop** ℓ' **determined by** $(\overrightarrow{\ell}, V)$ **from** M constructs a lens which removes column ℓ' from view M, given that the functional dependencies of the view ensure that ℓ' is determined by the columns $\overrightarrow{\ell}$. The typing rule is as follows:

T-Drop

$$\frac{\begin{array}{c}\mathcal{F} \equiv \mathcal{G} \cup \{\overrightarrow{\ell} \to \ell'\} \qquad \Gamma \vdash M : \textbf{lens of } (\Sigma, R \oplus (\ell' : A), \lambda x.\, P, \mathcal{F}) \\ \overrightarrow{\ell} \subseteq \mathsf{dom}(R) \qquad \Gamma \vdash V : A \qquad \mathbf{LJD}_{R,(\ell':A)}\ (\lambda x.\, P) \\ \mathbf{DV}_{R,(\ell':A)}\ (\lambda x.\, P)\ (\ell' = V) \qquad P' = P[V/x.\ell']\end{array}}{\Gamma \vdash \textbf{drop } \ell' \textbf{ determined by } (\overrightarrow{\ell}, V) \textbf{ from } M : \textbf{lens of } (\Sigma, R, \lambda x.\, \lambda x.P', \mathcal{G})}$$

The clause $\mathcal{F} \equiv \mathcal{G} \cup \{\overrightarrow{\ell} \to \ell'\}$ checks that the functional dependencies of the underlying lens M imply that $\overrightarrow{\ell}$ do indeed determine ℓ'; that $\overrightarrow{\ell}$ are contained in the domain of the record type R of underlying lens M; 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** $(\Sigma, R, \lambda x.\, P[V/x.\ell'], \mathcal{G})$ 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.

Lens Get. Finally we define typing rules for making use of relational lenses. Since Links 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.

T-Get

$$\frac{\Gamma \vdash M : \textbf{lens of } (\Sigma, R, \lambda x.\, P, \mathcal{F})}{\Gamma \vdash \textbf{get } M : \textbf{record set of } R}$$

Lens Put. Just as with T-Get, we have no way of statically ensuring that the input satisfies P and $\mathcal{F}$, so we only statically check that the updated view is a set of records matching type R, deferring the checks to ensure that the set of records satisfies $\mathcal{F}$ and P to runtime.

To ensure that the constraint P applies to each record r in a view, runtime checks ensure that $\mathsf{sat}(\lambda x.\, P, r)$. 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.

T-Put

$$\frac{R \vdash M : \textbf{lens of } (\Sigma, R, \lambda x.\, \lambda x.P, \mathcal{F}) \qquad \Gamma \vdash N : \textbf{record set of } R}{\Gamma \vdash \textbf{put } M \textbf{ with } N : ()}$$

3.4 Correctness

Bohannon et al. [5] prove that lenses satisfying correctness conditions are well-behaved (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 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. [5].

Figure 6 shows the grammar of sequential-style lenses. We let Π range over set-style predicates; S, T range over relation names; Σ, Δ range over schemas (i.e., sets of relation names); and I, J range over sequential-style lenses. The *sort* of a relation S, written sort (S) = $(\overrightarrow{\ell}, \Pi, \mathcal{F})$, is a 3-tuple of the set of fields $\overrightarrow{\ell}$ in S; a set predicate Π, and the set of functional dependencies $\mathcal{F}$. If sort $(S) = (\overrightarrow{\ell}, \Pi, \mathcal{F})$, then $\mathsf{dom}(S) = \overrightarrow{\ell}$.

Syntax of sequential lenses

Set predicates	Π, Π'	Schemas	Σ, Δ
Sequential lenses	$I, J ::=$		

$$\begin{array}{ll} & \mathbf{id} \mid I; J \\ \mid & \texttt{select from } S \texttt{ where } \Pi \texttt{ as } T \\ \mid & \texttt{join\_dl } S_1, S_2 \texttt{ as } T \\ \mid & \texttt{drop } \ell \texttt{ determined by } (\overrightarrow{\ell'}, V) \texttt{ from } S \texttt{ as } T \end{array}$$

Flattening translation $\boxed{(\!|M|\!) = \Sigma/I/S}$

$(\!|\textbf{lens } S \textbf{ with } \mathcal{F}|\!) = \{S\}/\mathbf{id}/S$

$(\!|\textbf{select}_{\lambda x.\, P} \textbf{ from } M|\!) =$
  $\Sigma/I; \texttt{select from } S \texttt{ where set}(\lambda x.\, P, \mathsf{dom}(S)) \texttt{ as } T/T$
 where $(\!|M|\!) = \Sigma/I/S$ and T is globally unique

$(\!|\textbf{join } M \textbf{ with } N \textbf{ delete\_left}|\!) =$
  $\Sigma \uplus \Delta/I; J; \texttt{join\_dl } S_1, S_2 \texttt{ as } T/T$
 where $(\!|M|\!) = \Sigma/I/S_1$, $(\!|N|\!) = \Delta/J/S_2$ and T is globally unique

$(\!|\textbf{drop } \ell \textbf{ determined by } (\overrightarrow{\ell'}, V) \textbf{ from } M|\!) =$
  $\Sigma/I; \texttt{drop } \ell \texttt{ determined by } (\overrightarrow{\ell'}, V) \texttt{ from } S \texttt{ as } T/T$
 where $(\!|M|\!) = \Sigma/I/S$

Figure 6: Sequential-style lenses [5] and flattening

Sequential-style lenses map source schemas to view schemas. The **id** lens defines the *identity* lens, mapping a schema to itself, and $I; J$ composes lenses I and J. The `select from` S `where` Π `as` T lens filters relation S using predicate set Π, 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.

Finally, `drop` ℓ `determined by` $(\overrightarrow{\ell'}, V)$ `from` S `as` T drops attribute ℓ determined by attributes $\overrightarrow{\ell'}$ with default value V from relation S, naming the resulting relation T.

Figure 6 also shows the translation from functional lenses to sequential-style lenses, which involves flattening functional lenses by introducing intermediate relations with fresh table names. The translation function $(\!|M|\!) = \Sigma/I/S$ states that functional lens M depends on tables Σ, translates to sequential lens I, and produces a view with name S.

As an example of a typing rule for sequential-style lenses, consider the typing rule for the select lens:

T-Select-RL

$$\frac{\begin{array}{c}\mathsf{sort}\,(S) = (\overrightarrow{\ell}, \Pi', \mathcal{F}) \qquad \mathsf{sort}\,(T) = (\overrightarrow{\ell}, \Pi \cap \Pi', \mathcal{F}) \\ \mathcal{F} \text{ is in tree form} \qquad \Pi' \text{ ignores outputs}(\mathcal{F})\end{array}}{\texttt{select from } S \texttt{ where } \Pi \texttt{ as } T \in \Sigma \uplus \{S\} \Leftrightarrow \Sigma \uplus \{T\}}$$

The sequential lens typing judgement has the shape $I \in \Sigma \Leftrightarrow \Delta$, meaning that I is a lens transforming the source schema Σ into the view schema Δ. In the case of the select lens, given a predicate set Π, the typing rule enforces the invariant that the source relation S has sort $(\overrightarrow{\ell}, \Pi', \mathcal{F})$; that the functional dependencies $\mathcal{F}$ are in tree form; that Π' ignores the outputs of $\mathcal{F}$; and assigns the view T the sort $(\overrightarrow{\ell}, \Pi \cap \Pi', \mathcal{F})$.

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. [5], and can use the incremental semantics described by Horn et al. [20].

Theorem 5 (Soundness of Translation). *If* $\Gamma \vdash M : \textbf{lens of}\ (\Sigma, R, \lambda x.\ P, \mathcal{F})$ *and* $(\!| M |\!) = \Sigma / L / T$*, then* $L \in \Sigma \Leftrightarrow \{T\}$ *and* $\mathrm{sort}(T) = (\mathrm{dom}(R), \mathrm{set}(\lambda x.\ P, R), \mathcal{F})$.

Proof. By induction on the derivation of $\Gamma \vdash M : \textbf{lens of}\ (\Sigma, R, \lambda x.\ P, \mathcal{F})$. □

3.5 Typechecking Dynamic Predicates

If a dynamic predicate is used in any lens combinator, the same checks are performed, but checking of predicates must be deferred to runtime. In this case, we require the programmer to acknowledge that the lens construction may fail at run-time. We introduce a special lens, the **check** lens, which the user must incorporate prior to using the lens in a **get** or **put** operation.

4 CASE STUDY: CURATED SCIENTIFIC DATABASES

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.

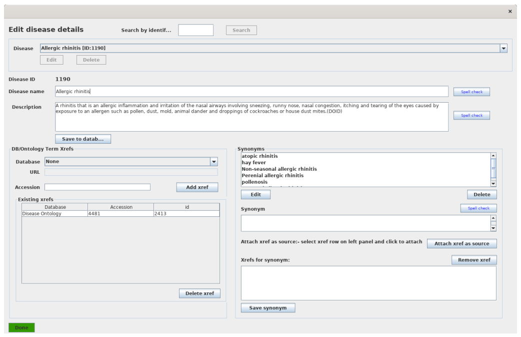
The IUPHAR/BPS Guide to Pharmacology (GtoPdb) [23] is a curated scientific database which collects information on pharmacological targets, such as receptors and enzymes, and *ligands* such as pharmaceuticals which act upon targets. GtoPdb consists of a PostgreSQL database, a Java/JSP web application frontend to the database, and a Java GUI application used for data curation.

In parallel work [15], we have implemented a workalike frontend application 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].

4.1 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 database occurs using relational lenses.

Figure 7a 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.



Disease: Allergic rhinitis [ID:1190]

Disease ID: 1190

Disease name: Allergic rhinitis

Description: A rhinitis that is an allergic inflammation and irritation of the nasal airways involving sneezing, runny nose, nasal congestion, itching and tearing of the eyes caused by exposure to an allergen such as pollen, dust, mold, animal dander and droppings of cockroaches or house dust mites.(DOID)

DB/Ontology Term Xrefs

Database: None

URL:

Accession:

Existing xrefs

Database	Accession	id
Disease Ontology	4481	2413

Synonyms

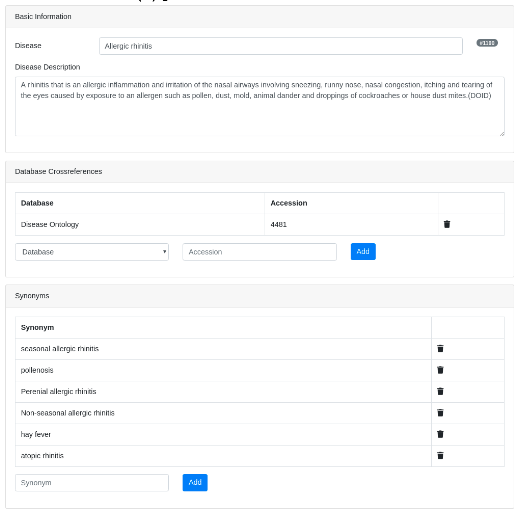
atopic rhinitis
hay fever
Non-seasonal allergic rhinitis
Perenial allergic rhinitis
pollenosis

Synonym:

Attach xref as source:- select xref row on left panel and click to attach

Xrefs for synonym:

(a) Java curation interface



Basic Information

Disease: Allergic rhinitis #1190

Disease Description

A rhinitis that is an allergic inflammation and irritation of the nasal airways involving sneezing, runny nose, nasal congestion, itching and tearing of the eyes caused by exposure to an allergen such as pollen, dust, mold, animal dander and droppings of cockroaches or house dust mites.(DOID)

Database Crossreferences

Database	Accession	
Disease Ontology	4481	🗑

Database | Accession

Synonyms

Synonym	
seasonal allergic rhinitis	🗑
pollenosis	🗑
Perenial allergic rhinitis	🗑
Non-seasonal allergic rhinitis	🗑
hay fever	🗑
atopic rhinitis	🗑

Synonym

(b) Links reimplementation

Figure 7: Curation interfaces for Diseases

4.2 Links Reimplementation

Figure 7b shows the curation interface as a Links web application. In the original implementation of Links [8], 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.

4.2.1 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.

4.2.2 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);
# 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);
# 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 [22] 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.

4.2.3 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-View-Controller 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.

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.

4.3 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.

5 RELATED WORK

Edit Lenses. Edit lenses are a form of bidirectional transformation where, rather than translating directly between one data structure and another, the changes to a data structure are tracked and then translated into changes to the other data structure [17]. They can be particularly useful in the case of *symmetric lenses* in situations where neither of the data structures contain all of the data, and thus none of the sources can be considered the 'source' [16]. 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.

Relational lenses are generally not considered edit lenses, as they directly translate the entire view to an updated source when performing get. *Incremental Relational Lenses* on the other hand take the updated view and compute a delta which is then translated into a delta to the source tables [20].

The language integration aspect of relational lenses is not dependent on the semantics used to perform relational updates. Instead it only relies on all of the relational lens typing rules in §3.3 to be satisfied; in this case, both the incremental and the non-incremental relational put semantics are guaranteed to be well-behaved.

Put-based Lenses. Bidirectional lenses are often defined in a form that corresponds to the forward (get) direction and the reverse direction. A common issue with this approach is that a get function might correspond to several well-behaved put functions, as illustrated by drop and join relational lenses. As such, defining a bidirectional transformation by only specifying the forward direction is generally not sufficient. An alternative approach recently used is to rather have the programmer instead only specify the *put* semantics, which then uniquely define the *get* semantics [13, 21].

A *putback* approach to bidirectional transformations has been recently proposed by Asano et al. [3] for relational data. Asano et al. define a language which allows the specification of update queries, for which the forward query can automatically be derived. They support splitting views vertically for defining behaviour specific to columns and horizontally for behaviour specific to rows. For each of the different sections of the view they can then define the update behaviour, which can be simple checks or actual update semantics.

Cross-tier web programming. SMLServer [12] was among the first functional frameworks to allow interaction with a relational database. Ur/Web [6] is a cross-tier web programming language which supports a statically-typed SQL DSL, along with atomic transactions and functional combinators on results. Neither framework supports language-integrated views.

Hop.js [28] builds on the Hop programming language [27] and allows cross-tier web programming in JavaScript. Eliom [26] is a cross-tier functional programming framework building on top of the OCaml programming language. Eliom programs can explicitly assign locations to functions and variables. ScalaLoci [29] is a Scala framework for cross-tier application programming. A key concept behind ScalaLoci is that data transfer between tiers uses the *reactive programming* paradigm. Haste.App [11] is a Haskell EDSL allowing web applications to be written directly in Haskell. Since these are embedded DSLs or frameworks, it becomes possible to use the database functionality provided by other libraries, but are not aware of any work providing relational lenses as a library in any programming language.

Task-oriented programming (TOP) [25] is a high-level paradigm centred around the idea of a *task*, which can be thought of as a unit of work with an observable value. TOP is implemented in the iTask system [24]. An *editor* is a task which interacts with a user. *Editlets* [10] are editors with customisable user interfaces, which can allow multiple users to interact with shared data sources. Much like incremental relational lenses [20], Editlets communicate *changes* in the data as opposed to the entire data source, however the user must specify this behaviour manually.

6 CONCLUSION

Relational lenses allow updatable views of database tables. Previous work has concentrated on the semantics of relational lenses, but has not proposed a concrete language design. As a result, previous implementations imposed severe limitations on predicates, and provided limited checking of the correctness of lens composition.

In this paper, we have presented the first full integration of relational lenses in a functional programming language, by extending the Links programming language. Building on the approach of Cooper [7], we use normalisation rules to rewrite functional expressions into a form amenable to compilation to SQL and for typechecking lenses. Furthermore, we have adapted the existing typing rules for relational lenses to the setting of a functional programming language and proved that our adapted rules are sound.

Previous implementations have provided only small example applications. To demonstrate the use of relational lenses, we have implemented part of the curation interface for a scientific database as a cross-tier web application, and shown how relational lenses can be used in tandem with the Model-View-Update architecture for frontend web development.

As future work, we plan to explore integrating relational lenses with auto-incrementing table fields.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for their helpful comments and to Simon Harding and Jo Sharman for discussion of the GtoPdb curation interface. This work was supported by ERC Consolidator Grant Skye (grant number 682315) and an ISCF Metrology Fellowship grant provided by the UK government's Department for Business, Energy and Industrial Strategy (BEIS).

REFERENCES

[1] 2019. Elm: A delightful language for reliable webapps. http://www.elm-lang.org.
[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases: the logical level.* Addison-Wesley Longman Publishing Co., Inc.
[3] Yasuhito Asano, Soichiro Hidaka, Zhenjiang Hu, Yasunori Ishihara, Hiroyuki Kato, Hsiang-Shang Ko, Keisuke Nakano, Makoto Onizuka, Yuya Sasaki, Toshiyuki Shimizu, et al. 2018. A View-based Programmable Architecture for Controlling and Integrating Decentralized Data. *arXiv preprint arXiv:1803.06674* (2018).
[4] François Bancilhon and Nicolas Spyratos. 1981. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 557–575.
[5] Aaron Bohannon, Benjamin C Pierce, and Jeffrey A Vaughan. 2006. Relational lenses: a language for updatable views. In *PODS*. ACM, 338–347.
[6] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *POPL*. ACM, 153–165.
[7] Ezra Cooper. 2009. The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. In *DBPL (Lecture Notes in Computer Science)*, Vol. 5708. Springer, 36–51.
[8] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web programming without tiers. In *FMCO*. Springer, 266–296.
[9] George Copeland and David Maier. 1984. Making smalltalk a database system. In *ACM Sigmod Record*, Vol. 14. ACM, 316–325.
[10] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Editlets: type-based, client-side editors for iTasks. In *IFL*. ACM, 6:1–6:13.
[11] Anton Ekblad. 2016. High-performance client-side web applications through Haskell EDSLs. In *Haskell*. ACM, 62–73.
[12] Martin Elsman and Niels Hallenberg. 2003. Web Programming with SMLserver. In *PADL (Lecture Notes in Computer Science)*, Vol. 2562. Springer, 74–91.
[13] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. 2015. The essence of bidirectional programming. *SCIENCE CHINA Information Sciences* 58, 5 (2015), 1–21.
[14] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007), 17.
[15] Simon Fowler, Simon Harding, Joanna Sharman, and James Cheney. 2020. Cross-tier web programming for curated databases: A case study. Under review.. http://arxiv.org/abs/2003.03845
[16] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2011. Symmetric lenses. In *POPL*, Vol. 46. ACM, 371–384.
[17] Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. 2012. Edit lenses. In *POPL*. ACM, 495–508.
[18] Rudi Horn, Simon Fowler, and James Cheney. 2020. Artifact for "Language-Integrated Updatable Views". https://doi.org/10.6084/m9.figshare.11907246
[19] Rudi Horn, Simon Fowler, and James Cheney. 2020. Language-Integrated Updatable Views (Extended version). https://arxiv.org/abs/2003.02191
[20] Rudi Horn, Roly Perera, and James Cheney. 2018. Incremental relational lenses. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 74.
[21] Zhenjiang Hu, Hugo Pacheco, and Sebastian Fischer. 2014. Validity checking of putback transformations in bidirectional programming. In *FM*. Springer, 1–15.
[22] Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *TLDI*. ACM, 91–102.
[23] Adam J. Pawson, Joanna L. Sharman, Helen E. Benson, Elena Faccenda, Stephen P.H. Alexander, O. Peter Buneman, Anthony P. Davenport, John C. McGrath, John A. Peters, Christopher Southan, Michael Spedding, Wenyuan Yu, Anthony J. Harmar, and NC-IUPHAR. 2013. The IUPHAR/BPS Guide to PHARMACOLOGY: an expert-driven knowledgebase of drug targets and their ligands. *Nucleic Acids Research* 42, D1 (11 2013), D1098–D1106.
[24] Rinus Plasmeijer, Peter Achten, and Pieter W. M. Koopman. 2007. iTasks: executable specifications of interactive work flow systems for the web. In *ICFP*. ACM, 141–152.
[25] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. 2012. Task-oriented programming in a pure functional language. In *PPDP*. ACM, 195–206.
[26] Gabriel Radanne, Vasilis Papavasileiou, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: tierless Web programming from the ground up. In *IFL*. ACM, 8:1–8:12.
[27] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: a language for programming the web 2.0. In *OOPSLA Companion*. ACM, 975–985.
[28] Manuel Serrano and Vincent Prunet. 2016. A glimpse of Hopjs. In *ICFP*. ACM, 180–192.
[29] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed system development with ScalaLoci. *PACMPL* 2, OOPSLA (2018), 129:1–129:30.
[30] Limsoon Wong. 2000. Kleisli, a functional query system. *Journal of Functional Programming* 10, 1 (2000), 19–56.