Language-Integrated Updatable Views
Extended version

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. [19] 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.

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 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, 29]. 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 get AlbumsByName(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"
```

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
Relational Lenses. A recent approach to the view update problem is to define views using compositional 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:

$$\text{GetPut } \text{put } \ell (s) = s \quad \text{PutGet } \text{get } (\text{put } s \; \ell) = v$$

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. [19] recently presented the first implementation using an incremental semantics. However Horn et al. [19] 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 $R \rightarrow \text{bool}$ 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:

```plaintext
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.

```plaintext
fun getAlbumsByNameL(albumName) {
  var albumLens = lens albums where album -> quantity;
  var selectLens = select from albumLens 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 $\ell \rightarrow v'$ states that the columns in $\ell'$ are uniquely determined by $\ell$; 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
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:

```plaintext
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.

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.

| Syntax          | Types      | Base types | Base record types |
|-----------------|------------|------------|-------------------|
|                 | A, B, C    | D          | R                 |
|                 | A → B      | bool | int | string |
|                 | (fun : A)  |            |                   |

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.
Normal forms

\[
O ::= x \mid c \mid \lambda x.O \mid (\ell = \overrightarrow{O}) \mid x.\ell
\mid \text{if } O_1 \text{ then } O_2 \text{ else } O_3 \mid \varnothing (\overrightarrow{O})
\]

\[
P, Q ::= \text{if } P_1 \text{ then } P_2 \text{ else } P_3 \mid \varnothing (P) \mid x.\ell \mid c
\]

Normalisation

\[
(M : A \rightarrow N) \rightarrow (xN) M : A \rightarrow N[M/x]
\]

\[
(\ell = M), \ell : A \rightarrow M\ell
\]

\[
\text{if true then } L \text{ else } M : A \rightarrow M
\]

\[
\text{if false then } L \text{ else } M : A \rightarrow M
\]

\[
\text{if } L \text{ then } M \text{ else } M' : (\ell : A) \rightarrow (\ell = \overrightarrow{N})
\]

\[
\begin{aligned}
&\text{if } L \text{ then } M \text{ else } M', \ell \\
&\text{for each } \ell \in I
\end{aligned}
\]

Evaluation

\[
V ::= c \mid \lambda x.M \mid (\ell = \overrightarrow{V})
\]

\[
\begin{aligned}
&M \upharpoonright V \\
&M \upharpoonright V \upharpoonright W \\
&M \upharpoonright (\ell = \overrightarrow{V})
\end{aligned}
\]

Figure 4: Normalisation and Evaluation

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, \(\lambda\)-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 \(P\), are a restriction of terms in normal form. Terms in predicate normal form have a straightforward SQL equivalent, and can be used when typechecking lenses.

Normalisation rules \(M \rightarrow N\) are a subset of the rules proposed by Cooper [7]: 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 1 (Strong normalisation). If \(\Gamma \vdash M : A\), then there are no infinite \(\rightarrow\) 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 \(\rightarrow\) 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 \(M \rightarrow\) \(N \vdash\), then \(N\) is in normal form.

Proof. By induction on the derivation of \(x : R \vdash M : A\). The details can be found in Appendix B.

As a corollary, by considering only terms with type \(\text{bool}\), we can show that static predicates are in predicate normal form.

Corollary 3 (Predicate normal form). If \(x : R \vdash M : \text{bool}\) and \(M \rightarrow\) \(N \vdash\), 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 \(M \upharpoonright V\), which states that term \(M\) evaluates to a value \(V\). We use the notation \(\varnothing(V)\) to denote the denotation of operation \(\varnothing\) applied to arguments \(V\); for example, \(\varnothing[5,10] = 15\). The semantics enjoys a standard type soundness property.

Proposition 4 (Type Soundness). If \(\cdot \vdash M : A\), then there exists some \(V\) such that \(M \upharpoonright V\) 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 \(\overrightarrow{\ell} \rightarrow R\) requires that two records with the same values for \(\overrightarrow{\ell}\) should have the same values for \(R\). 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 \(\mathcal{F} \vdash \mathcal{F} \rightarrow \mathcal{F}'\) specifies that the functional dependency \(\mathcal{F} \rightarrow \mathcal{F}'\) can be derived from the set of functional dependencies \(\mathcal{F}\) following Armstrong’s axioms [2]; these (standard) derivation rules can be found in Appendix A. The output fields of the functional dependencies \(\mathcal{F}\), written \(\text{outputs}(\mathcal{F})\), is the set of fields constrained by \(\mathcal{F}\) and is defined as:

Definition 1 (Output fields).

\[\text{outputs}(\mathcal{F}) = \{\ell \in \overrightarrow{\ell} \mid \exists \overrightarrow{\ell}' \in \overrightarrow{\ell}. \ell \not\in \overrightarrow{\ell}' \text{ and } \mathcal{F} \models \overrightarrow{\ell}' \rightarrow \ell\}\]
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, \( A \rightarrow B, A \rightarrow C, C \rightarrow D \) 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, \( \{ A \rightarrow BC, C \rightarrow D \} \) 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 \( \text{table of } (S, R) \) is the type of a table with table name \( S \) containing records of type \( R \). The record set type \( \text{record set of } R \) describes a set of records of type \( R \). The type of lenses, \( \text{lens of } (S, R, \lambda x. P, \exists \mathcal{J}) \), consists of four components: the set of underlying tables \( S \); the base record type \( R \); a restriction predicate \( \lambda x. P \); and a set of functional dependencies \( \mathcal{J} \). 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 \( r @ s = (\ell_1 = V_1, \ldots, \ell_n = V_n) \).
- 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 \( R @ R' = (\ell_1 : A_1, \ldots, \ell_n : A_n) \).

**Tables.** Links defines a primitive table expression \( \text{table } S \text{ with } R \) 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.

\[
\Gamma \vdash \text{table } S \text{ with } R : \text{table of } (S, R)
\]

**Lenses Primitives.** The rule \( \text{T-Lens} \) is used to create a relational lens from a Links table. A lens primitive is assigned the default predicate constraint \( \text{true} \). All columns referred to by a set of functional dependencies \( \mathcal{J} \), written names(\( \mathcal{J} \)), should be part of the table record type \( R \).

\[
\begin{align*}
\Gamma & \vdash M: \text{table of } (S, R) \quad \bigcup \text{names(} \mathcal{J} \text{)} \subseteq \text{dom}(R) \\
\Gamma & \vdash \text{lens of } (S, R, \lambda x. \mathcal{J}, \exists \mathcal{J})
\end{align*}
\]

#### 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 |
|---------|------|--------|-------|----------|
| Lovesong| 1989 | 4      | Galore| 1        |

The user then notices that the rating for Lovesong is not correct, and changes it from 5 to 4. Calling \( \text{put} \) on \( l_3 \) would yield the updated view for \( l_2 \):

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

**Definition 3 (Predicate Ignores).** We say \( P \) ignores \( \ell \) if there exists an \( R \) such that \( \ell \) is disjoint from \( \text{dom}(R) \) and \( x : R \vdash P : \text{bool} \).
The T-Select rule also needs to ensure that the resulting lens only accepts records that satisfy the given predicate \( \lambda x. P \) 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 \( \lambda x. P \land Q \). The full select lens typing rule can be defined as:

\[
\Gamma \vdash M : \text{lens of } \left( \Sigma, R, \lambda x. P, \scr{F} \right) \quad x : R \vdash Q : \text{bool} \\
\scr{F} \text{ is in tree form} \\
P \text{ ignores outputs}(\scr{F}) \\
\Gamma \vdash \text{select } x, Q \text{ from } M : \text{lens of } \left( \Sigma, R, \lambda x. P \land Q, \scr{F} \right)
\]

### 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`.

| 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 |
| thecure | 5      | Trust   | 1992 | 4      | Wish  |

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 \( \tau \cap \tau' \rightarrow \overset{\downarrow}{\tau} \), where \( \tau \cap \tau' \) are the join columns and \( \overset{\downarrow}{\tau} \) 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 \( \tau \cap \tau' \) and then checking if \( \overset{\downarrow}{\tau} \) 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(\( \tau \)) and \( Q \) ignores outputs(\( \tau' \)). The resulting lens should have the predicate \( P \land Q \) since the record constraints of both input lenses apply to the output lens.

---

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 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 \( \Pi, \Pi' \) 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 \( \Pi \) to \( \{\ell\} \), written \( \Pi[\{\ell\}] \), as \( \{r[\{\ell\}] \mid r \in \Pi\} \).

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 \( \Pi \) contains records of type \( R_1 \) and \( \Pi' \) contains records of type \( R_2 \).

We define the set join of \( \Pi \) and \( \Pi' \), written \( \Pi \bowtie \Pi' \), as:
\[
\{r \in \text{inh}(R) \mid \exists s \in \Pi, \exists t \in \Pi' \text{ such that } r[\text{dom}(R_1)] = s[\text{dom}(R_2)]\}
\]

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 \( x : R_1 @ R_2 \vdash P : \text{bool} \), written \( \text{LJD}_{R_1,R_2}(\lambda x. P) \), means that for all \( r_1, r_2 \in \text{inh}(R_1) \) and \( s_1, s_2 \in \text{inh}(R_2) \), it is the case that:
\[
\text{sat}(\lambda x. P, r_1 \oplus s_1) \wedge \text{sat}(\lambda x. P, r_2 \oplus s_2) \implies \text{sat}(\lambda x. P, r_1 \oplus s_2)
\]

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

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

Proof. Follows from the definitions of \( \text{LJD}_{R_1,R_2}(\lambda x. P) \), \( \cdot \bowtie \cdot \) and \( \cdot \vdash \cdot \). For further details see Appendix C.3.4. \( \square \)

Showing \( \text{LJD}_{R,R'}(\lambda x. P) \) is NP-hard and could be undecidable, depending on the atomic formulae available in the predicates. Since a predicate that satisfies \( \text{LJD}_{R,R'}(\lambda x. P) \) 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 \( \text{LJD}_{R,R'}(\lambda x. P) \).

\[
\begin{align*}
\text{LJD}_{R,R'}(\lambda x. P) & \quad \text{LJD}_{R,R'}(\lambda x. Q) \\
\text{LJD}_{R,R'}(\lambda x. P) & \quad \text{LJD}_{R,R'}(\lambda x. P \land Q)
\end{align*}
\]

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

Proof. By induction on the derivation of \( \text{LJD}_{R,R'}(\lambda x. P) \). \( \square \)

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 set(\( \lambda x. P, R \)), we must show that \( \{V = V\} \subseteq \text{set}(\lambda x. P, R)[\{\ell\}'] \).

We define a property \( \text{DV}_{R,R'}(\lambda x. P) \) 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 \( \text{LJD}_{R,R'}(\lambda x. P) \), and \( r \in \text{inh}(R') \), we write \( \text{DV}_{R,R'}(\lambda x. P) \) when \( \text{set}(\lambda x. P, R \oplus R') \) is not empty and there exists an \( s \in \text{inh}(R) \) such that \( \text{sat}(\lambda x. P, r \oplus s) \).

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

Proof. By expansion of the definitions of \( \text{DV}_{R,R_1}(\lambda x. P) \) and of \( \cdot \oplus \cdot \) and \( \cdot \vdash \cdot \). For details see Appendix C.3.4. \( \square \)

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

\[
\begin{align*}
\text{DV}^1_{R,R'}(\lambda x. P) & \quad \text{DV}^1_{R,R'}(\lambda x. P) \\
\text{DV}^1_{R,R'}(\lambda x. P) & \quad \text{DV}^1_{R,R'}(\lambda x. P) \land \text{DV}^1_{R,R'}(\lambda x. Q)
\end{align*}
\]

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

Proof. By induction on the derivation of \( \text{DV}^1_{R,R'}(\lambda x. P) \). For further details see Appendix C.3.4. \( \square \)

Note that the soundness proof for \( \text{DV}^1_{R,R'}(\lambda x. P) \) requires that \( \text{set}(\lambda x. P, R \oplus R') \) 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 \textit{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 \textit{drop} \( \ell' \) determined by \( (\ell, V) \) from \( M \) constructs a lens which removes column \( \ell' \) from view \( M \), given that the functional dependencies of the view ensure that \( \ell' \) is determined by the columns \( \ell \). The typing rule is as follows:
well-behavedness. Our approach is to define a type-preserving style lenses defined by Bohannon et al. [5].

The clause \( \vdash \) \((\ell', V)\) from \(M: \text{len}\) of \((\Sigma, R, \lambda x. P, \mathcal{T})\) checks that the functional dependencies of the underlying lens \(M\) imply \(\mathcal{T}\) do indeed determine \(\ell'\); that \(\mathcal{T}\) 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 \(\text{len}\) of \((\Sigma, R, \lambda x. P[V/x, \ell'], \mathcal{S})\) 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.

### 3.4 Correctness

Bohannon et al. [5] prove that lenses satisfying correctness conditions are well-behaved (i.e., satisfy \(\text{GETPut}\) and \(\text{PUTGET}\), 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 \(\Pi\) range over set-style predicates; \(S, T\) range over relation names; \(\Sigma, \Delta\) range over schemas (i.e., sets of relation names); and \(I, J\) range over sequential-style lenses. The sort of a relation \(S\), written \(\text{sort}(S) = (\mathcal{T}, \Pi, \mathcal{S})\), is a 3-tuple of the set of fields \(\mathcal{T}\) in \(S\); a set predicate \(\Pi\), and the set of functional dependencies \(\mathcal{F}\). If \(\text{sort}(S) = (\mathcal{T}, \Pi, \mathcal{F})\), then \(\text{dom}(S) = \mathcal{T}\).

Sequential-style lenses map source schemas to view schemas. The \(\text{id}\) lens defines the identity lens, mapping a schema to itself, and \(I, J\) composes lenses \(I\) and \(J\). The \select{} from \(S\) where \(\Pi\) as \(T\) lens filters relation \(S\) using predicate set \(\Pi\), naming the resulting relation \(T\). The \joint{} \(S_1, S_2\) as \(T\) lens joins relations \(S_1\) and \(S_2\) using the delete-left strategy, naming the resulting relation \(T\).

Finally, \(\text{drop} \ \ell\ \text{determined by} \ (\mathcal{T}, V)\ \text{from} \ S\ \text{as} \ T\) drops attribute \(\ell\) determined by attributes \(\mathcal{T}\) 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 \(\langle M \rangle = \Sigma/I/S\) states that functional lens \(M\) depends on tables \(\Sigma\), 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:

### 3.5 Syntax of sequential lenses

- Set predicates \(\Pi, \Pi'\)
- Schemas \(\Sigma, \Delta\)
- Sequential lenses \(I, J\)

### 3.6 Flattening translation

\[ \langle M \rangle = \Sigma/I/S \]
form; that $\Pi'$ ignores the outputs of $\mathcal{F}$; and assigns the view $T$ the sort $(\mathcal{T}, \Pi \cap \Pi', \mathcal{T})$.

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. [19].

**Theorem 5 (Soundness of Translation).**

If $\Gamma \vdash M : \text{Lens of } (\Sigma, R, \lambda x. P, \mathcal{F})$ and $\{M\} = \Sigma/L/T$, then $L \in \Sigma \Leftrightarrow \{T\}$ and $\text{sort}(T) = (\text{dom}(R), \text{set}(\lambda x. P, R), \mathcal{F})$.

**Proof.** By induction on the derivation of $\Gamma \vdash M : \text{Lens of } (\Sigma, R, \lambda x. P, \mathcal{F})$; see Appendix C. □

### 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) [22] 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 interfaces respectively.

#### 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.

```haskell
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).

```haskell
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, name: String,
   description: String, type: 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:

```haskell
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.

```haskell
fun diseaseFilter(x) { x.disease_id == diseaseID }
```

```haskell
# 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);
```

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.

```haskell
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 [21] 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.
### 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.

```haskell
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.

```haskell
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
    }
}
```

### 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 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 [19].

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, 20].

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 [27] builds on the Hop programming language [26] and allows cross-tier web programming in JavaScript. Eliom [25] 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 [28] 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) [24] 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 [23]. 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 [19], 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 GoToPdb curation interface. This work was supported by ERC Consolidator Grant Skye (grant number 682315) and an ISCF MetroFellowship 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] Yasunari Asano, Soichiro Hidada, Zhenjiang Hu, Yasuoori Ishihara, Hiroyuki Kato, Hsiang-Shong Ko, Keisuke Nakano, Makoto Onzuka, 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 Bancillon and Nicolas Spreitzer. 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] Erza Cooper. 2009. The Script-Writer’s Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. In DBLP (Lecture Notes in Computer Science). Vol. 5709. Springer, 36–51.
[8] Erza Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web programming without tiers. In IFLC. 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 Lijnze, and Rinus Plasmeijer. 2014. Editlets: type-based, client-side editors for iTasks. In IFL. ACM, 6:1–6:13.
[11] Anton Ekkblad. 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 PPLD (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.
[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, Roly Perera, and James Cheney. 2018. Incremental relational lenses. Proceedings of the ACM on Programming Languages 2, ICFP (2018), 74.
[20] Zhenjiang Hu, Hugo Pacheco, and Sebastian Fischer. 2014. Validity checking of putback transformations in bidirectional programming. In FM. Springer, 1–15.
[21] Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In TLDI. ACM, 91–102.
[22] Adam J. Pawson, Joanna L. Sharman, Helen E. Benson, Elena Facenda, Stephen P.H. Alexander, O. Peter Buneman, Anthony P. Davenport, John C. McGrath, John A. Peters, Christopher Southan, Michael Spedding, Wenjun 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 (2013), D1098–D1106.
[23] 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.
[24] Rinus Plasmeijer, Bas Lijnze, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. 2007. Task-oriented programming in a pure functional language. In PPDP. ACM, 195–206.
[25] Gabriel Radanne, Vasilis Papavasileiou, Jérôme Vouillon, and Vincent Balat. 2016. A View-based Programmable Architecture for Controlling and Integrating: Decentralized Data. arXiv preprint arXiv:1803.06674 (2018).
[26] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: a language for cross-tier functional programming framework building on top of the OCaml programming language. Eliom programs can explicitly assign locations to functions and variables. ScalaLoci [28] 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) [24] 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 [23]. 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 [19], 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 GoToPdb curation interface. This work was supported by ERC Consolidator Grant Skye (grant number 682315) and an ISCF MetroFellowship grant provided by the UK government’s Department for Business, Energy and Industrial Strategy (BEIS).
A SUPPLEMENTARY MATERIAL

A.1 Functional Dependencies

Figure 8 defines the set of functional dependencies which can be derived from a set of functional dependencies $\mathcal{F}$.

Functional Dependencies

$$F, G ::= \ell \rightarrow \ell'$$

Armstrong’s Axioms

- **FD-ID**
  $\ell \rightarrow \ell' \in \mathcal{F}$

- **FD-Transitive**
  $\mathcal{F} \models \ell \rightarrow \ell'$$ $\mathcal{F} \models \ell' \rightarrow \ell''$

- **FD-Reflexive**
  $\ell \rightarrow \ell \in \mathcal{F}$

- **FD-Aug**
  $\mathcal{F} \models \ell \rightarrow \ell'$$ \mathcal{F} \models \ell' \rightarrow \ell''$

- **FD-Composition**
  $\mathcal{F} \models \ell \rightarrow \ell'$$ \mathcal{F} \models \ell' \rightarrow \ell''$$ \mathcal{F} \models \ell \rightarrow \ell''$

- **FD-Decomposition**
  $\mathcal{F} \models \ell \rightarrow \ell'$$ \mathcal{F} \models \ell' \rightarrow \ell''$

Figure 8: Armstrong’s functional dependency axioms.

B PROOFS FOR SECTION 2

**Proposition 2** If $x : R \vdash M : A$ and $M \not\Rightarrow^* N \not\Rightarrow$, then $N$ 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:

$$x : R \vdash M : A \rightarrow B$$

By the induction hypothesis, we have that $M$ and $N$ are in normal form.

Given that $M$ is in normal form and has function type, there are the following possibilities:

- $M = x$, which is not possible since the only variable in the typing environment is $x$, and $R$ is not a function type
- $M = c$, which is not possible since constants only have base types, not function types
- $M = \lambda x. M$. In this case, we could apply the first normalisation rule, which would be a contradiction.
- $M = x. \ell$, which is not possible since $R$ only contains fields with base types, not function types
- $\text{if } V_1 \text{ then } V_2 \text{ else } V_3$. In this case, we could apply the fifth normalisation rule, which would be a contradiction.

Thus, a term $x : R \vdash M N$ cannot be in normal form.

**Case** $T$-Project

Assumption:

$$\Gamma \vdash M : (\ell_i : A_i)_{i \in I}$$

By the IH, we have that $M$ is in normal form. We now perform case analysis on $M$, giving us the following possibilities for terms in normal form which can have record type:

- $M = x$: We have that $x. \ell$ which is in normal form.
- $M = (\ell \equiv V)$: Impossible, since it would be possible to reduce by the second normalisation rule
- $M = \text{if } V_1 \text{ then } V_2 \text{ else } V_3$: Impossible, since it would be possible to reduce by the sixth reduction rule
The following definition for set ignores has been taken from Bohannon et al. [5]:

**Definition 12.** set\(\lambda x. P, R \otimes R'\) ignores \(R'\) if for all \(r, s\) if \(\forall r [\text{dom}(R)] = s[\text{dom}(R)]\) then \(r \in \text{set}(\lambda x. P, R \otimes R') \iff s \in \text{set}(\lambda x. P, R \otimes R').\)

**Lemma 5 (Set-Ignores).** Suppose \(P\) ignores \(\text{dom}(R)\). Then set\((\lambda x. P, R)\) ignores \(\text{dom}(R')\).

**Proof.**

\[\Gamma \vdash L : \text{bool} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A\]

\[\Gamma \vdash \text{if } L \text{ then } M \text{ else } N : A\]

Immediate by the induction hypothesis on all three subterms; normalisation rules 3 and 4 serve only as an optimisation.

\[\square\]

## C PROOFS FOR SECTION 2

### C.1 Predicate Lemmas

The following definition for set ignores has been taken from Bohannon et al. [5]:

**Lemma 6.** Suppose two disjoint type contexts \(R\) and \(R'\) as well as \(r \in \text{inh}(R)\) and \(s \in \text{inh}(R')\). Then sat\((\lambda x. P, r \otimes s)\) implies \(r \in \{z[R] \mid z \in \text{inh}(R \otimes R')\} \text{ and } s \in \{z[R'] \mid z \in \text{inh}(R \otimes R')\} \text{ sat}(\lambda x. P, z)\) and

**Proof.** sat\((\lambda x. P, r_1 \otimes r_2)\) implies that there exists a \(s \in \text{inh}(R \otimes R')\) which is equal to \(r_1 \otimes r_2\) such that sat\((\lambda x. P, s)\). We know that for this \(s\), \(s[\text{dom}(R_1)]\) equals \(r_1\) by definition of \(\cdot \otimes \cdot\). Conversely the same can be shown for \(r_2\).

**Lemma 7.** \(r \in \text{set}(\lambda x. P, R)\) if and only if sat\((\lambda x. P, r)\).

**Proof.** By definition of sat\((\lambda x. P, \cdot)\), \(\cdot \in \cdot\) and set\((\lambda x. \cdot, \cdot)\).

### C.2 Predicate Equivalences

**Lemma 8.** Suppose two predicates \(P, Q\) such that \(x : R \vdash P\) and \(x : R \vdash Q\). Then set\((\lambda x. P \wedge Q, R) = \text{set}(\lambda x. P, R) \cap \text{set}(\lambda x. Q, R)\).

**Proof.**

\[x : R \vdash P\]

\[\text{assumption}\]

\[x : R \vdash Q\]

\[\text{assumption}\]

\[\text{set}(\lambda x. P \wedge Q, R)\]

\[= \{r \mid \forall r \in \text{inh}(R). \text{sat}(\lambda x. P \wedge Q, r)\}\]

\[= \{r \mid \forall r \in \text{inh}(R). \text{sat}(\lambda x. P, r) \text{ and } \text{sat}(\lambda x. Q, r)\}\]

\[= \{r \mid \forall r \in \text{inh}(R). r \in \text{set}(\lambda x. P, R) \text{ and } s \in \text{set}(\lambda x. Q, R)\}\]

\[= \text{set}(\lambda x. P, R) \cap \text{set}(\lambda x. Q, R)\]

\[\text{def. } \cdot \wedge \cdot\]

\[\text{Lemma 7}\]

\[\text{def. } \cdot \cap \cdot\]

\[\text{def. set}(\lambda x. \cdot, \cdot)\]

\[\text{def. sat}(\lambda x. \cdot, \cdot)\]
C.3 Lens Translation

3.1 Primitive Lens.

Lemma 11. Suppose \( \Gamma \vdash \text{lens } M \text{ with } \mathcal{F} : \text{lens of } (\Sigma, \lambda x. \text{true}, \mathcal{F}) \). Then \([\text{lens } M \text{ with } \mathcal{F}] = \Sigma/\text{id}/\mathcal{F}\) and \(\text{sort } (T) = (\text{dom } (R), \text{set } (\lambda x. \text{true}, R), \mathcal{F})\).

Proof.

\[\begin{align*}
\Gamma \vdash M : \text{table of } (S, R) & \quad \bigcup \{\text{names } (\mathcal{F}) \subseteq \text{dom } (R)\} \\
\Gamma \vdash \text{lens } M \text{ with } \mathcal{F} : \text{lens of } (\{S\}, R, \lambda x. \text{true}, \mathcal{F}) & \quad \text{assumption}
\end{align*}\]

sort \((T) = (\text{dom } (R), \text{set } (\lambda x. \text{true}, R), \mathcal{F})\) define

\[\begin{align*}
\text{T-Lens} & \\
\text{sort } (R) = (U, \Pi, \mathcal{F}) & \quad \text{sort } (S) = (V, \Pi', \mathcal{F}) \\
\text{sort } (T) = (U \cup V \rightarrow V) & \quad \mathcal{G} \vdash U \cup V \rightarrow V \\
\Pi \text{ is in tree form} & \quad \mathcal{F} \text{ is in tree form} \\
\mathcal{G} \text{ ignores outputs } (\mathcal{F}) & \quad \Pi' \text{ ignores outputs } \mathcal{F} \\
\Pi = \pi (U - A) \equiv \Pi (A) & \quad \{A = a\} \in \Pi (A) \\
I \in \Sigma \equiv \Sigma' & \quad J \in \Sigma' \equiv \Delta \\
\text{T-Compose-RL} & \quad \text{T-Id-RL}
\end{align*}\]

Figure 9: Lens typing rules as defined by Bohannon et al. [5].
\[ L = \text{id} \in \{ S \} \Leftrightarrow \{ S \} \quad \text{T-Id-RL} \]

\[ \langle \text{lens } M \text{ with } \mathcal{F} \rangle = \{ S \}/L/S \]

\[ \Box \]

### C.3.2 Select Lens

**Lemma 12.** Suppose \( \Gamma \vdash \text{select}_{x \cdot \mathcal{Q}} \mathcal{M} \text{ of } (\Sigma, R, \lambda x. P \land Q, \mathcal{F}) \) and \( \langle \mathcal{M} \rangle = \Sigma/L/S \) such that \( L \in \Sigma \Leftrightarrow \{ S \} \) and sort \( S = (\text{dom}(R), \text{set}(\lambda x. P, R), \mathcal{F}) \). Then \( \langle \text{select}_{x \cdot \mathcal{Q}} \mathcal{M} \rangle = \Sigma/L'/T \) such that \( L' \in \Sigma \Leftrightarrow \{ T \} \) and sort \( T = (\text{dom}(R), \text{set}(\lambda x. P \land Q, R), \mathcal{F}) \).

**Proof.**

\[ \Gamma \vdash \text{M: lens of } (\Sigma, R, \lambda x. P \land Q, \mathcal{F}) \]

\[ \Gamma \vdash \text{x: R \vdash Q: bool} \]

\( \mathcal{F} \) is in tree form \quad \( P \) ignores outputs(\( \mathcal{F} \)) \quad \text{assumption (1)}

\[ \Gamma \vdash \text{select}_{x \cdot \mathcal{Q}} \mathcal{M} \text{ from } M: \text{lens of } (\Sigma, R, \lambda x. P \land Q, \mathcal{F}) \]

\[ \| \mathcal{M} \| = \Sigma/L/S \]

\( L \in \Sigma \Leftrightarrow \{ S \} \quad \text{assumption (2)}

sort \( S = (\text{dom}(R), \text{set}(\lambda x. P, R), \mathcal{F}) \)

\( \text{assumption (3)} \)

sort \( T = (\text{dom}(R), \text{set}(\lambda x. P \land Q, R), \mathcal{F}) \)

\( \text{define (5)} \)

\[ \sigma(\lambda x. Q, R) \text{ ignores outputs}(\mathcal{F}) \quad \text{Lemma 8 (6)} \]

\[ \sigma(\lambda x. Q, R) \text{ ignores outputs}(\mathcal{F}) \]

\[ \text{T-Select-RL (4, 6, 7) \quad (8)} \]

\[ L_x = \text{select from } S \text{ where } \sigma(\lambda x. Q, R) \text{ as } T \in \{ S \} \Leftrightarrow \{ T \} \]

\[ \text{T-Select-RL (4, 6, 7) \quad (8)} \]

\[ \langle \text{select}_{x \cdot \mathcal{Q}} \mathcal{M} \rangle = \Sigma/L; L_x/T \]

\[ \text{T-Compose-RL (3, 8) \quad \Box} \]

### C.3.3 Join Lens

**Lemma 13.** Suppose \( \Gamma \vdash \text{join } M \text{ with } N \text{ delete left: lens of } (\Sigma \cup \Delta, R \cup R', \lambda x. P \land Q, \mathcal{F} \cup \mathcal{G}) \) and \( \langle \mathcal{M} \rangle = \Sigma/L/S_1 \) and \( \langle \mathcal{N} \rangle = \Delta/L'/S_2 \) such that \( L_1 \in \Sigma \Leftrightarrow \{ S_1 \} \), sort \( S_1 = (\text{dom}(R), \text{set}(\lambda x. P, R), \mathcal{F}) \) and sort \( S_2 = (\text{dom}(R'), \text{set}(\lambda x. Q, R'), \mathcal{G}) \). Then \( \langle \text{join } M \text{ with } N \text{ delete left} \rangle = \Sigma \cup \Delta/L/T \) such that \( L \in \Sigma \cup \Delta \Leftrightarrow \{ T \} \) and sort \( T = (\text{dom}(R \oplus R'), \text{set}(\lambda x. P \land Q, R \oplus R'), \mathcal{F} \cup \mathcal{G}) \).

**Proof.**

\[ \text{T-Join-Left} \]

\[ \Gamma \vdash \text{M: lens of } (\Sigma, R, \lambda x. P, \mathcal{F}) \quad \Gamma \vdash \text{N: lens of } (\Delta, R', \lambda x. Q, \mathcal{G}) \]

\[ \Sigma \vdash \text{dom}(R) \land \text{dom}(R') \rightarrow \text{dom}(L') \quad \mathcal{F} \text{ is in tree form} \quad \mathcal{G} \text{ is in tree form} \quad \text{P ignores outputs}(\mathcal{F}) \quad \mathcal{Q} \text{ ignores outputs(\mathcal{G})} \quad \Sigma \cap \Delta = \emptyset \quad \text{assumption (1)} \]

\[ \langle \text{join } M \text{ with } N \text{ delete left: lens of } (\Sigma \cup \Delta, R \cup R', \lambda x. P \land Q, \mathcal{F} \cup \mathcal{G}) \]

\[ \| \mathcal{M} \| = \Sigma/L_1/S_1 \quad \text{assumption (2)} \]

\[ \| \mathcal{N} \| = \Delta/L_2/S_2 \quad \text{assumption (3)} \]

\( L_1 \in \Sigma \Leftrightarrow \{ S_1 \} \quad \text{assumption (4)} \)

\( L_2 \in \Delta \Leftrightarrow \{ S_2 \} \quad \text{assumption (5)} \)

\( \sigma(\lambda x. S_1, P, R) \)

\( \text{assumption (6)} \)

\( \sigma(\lambda x. S_2, Q, R') \)

\( \text{assumption (7)} \)

\( L_1 \in \Sigma \cup \Delta \Leftrightarrow \{ S_1 \} \cup \Delta \quad \text{weakening* (8)} \)

\( L_2 \in \{ S_1 \} \cup \Delta \Leftrightarrow \{ S_1, S_2 \} \quad \text{weakening* (9)} \)

\( L_1, L_2 \in \Sigma \cup \Delta \Leftrightarrow \{ S_1, S_2 \} \quad \text{T-Compose-RL} \)

\* all intermediate views are globally unique due to def. of \( \| \| \).

\[ \text{sort}(T) = (\text{dom}(R \oplus R'), \text{set}(\lambda x. P \land Q, R \oplus R'), \mathcal{F} \cup \mathcal{G}) \quad \text{define (10)} \]
Language-Integrated Updatable Views

\[ (\text{dom}(R) \oplus \text{dom}(R'), \text{set}(\lambda x. P, R)) \Rightarrow \text{set}(\lambda x. Q, R'), \mathcal{F} \cup \mathcal{S}) \]

Lemma 9 (11)

\[ \text{set}(\lambda x. P, R) \text{ ignores outputs(}\mathcal{F}) \]

Set-Ignores (1) (12)

\[ \text{set}(\lambda x. Q, R') \text{ ignores outputs(}\mathcal{S}) \]

Set-Ignores (1) (13)

\[ \text{LJD} = \text{join dt} S_1, S_2 \text{ as } T \in \{S_1, S_2\} \iff \{T\} \]

T-JOIN-RL

C.3.4 Drop Lens. Lemma 1 Suppose \( R = R_1 \oplus R_2 \) and \( x : R \vdash P : \text{bool} \). If \( \text{LJD}_{R_1, R_2} (\lambda x. P) \), then \( \text{set}(\lambda x. P, R) = \text{set}(\lambda x. P, R)[\text{dom}(R_1)] \cong \text{set}(\lambda x. P, R)[\text{dom}(R_2)] \).

Proof.

\[ R = R_1 \oplus R_2 \]

assumption

\[ \text{LJD}_{R_1, R_2} (\lambda x. P) \]

assumption

\[ \forall r, s \in \text{inh}(R), \text{sat}(\lambda x. P, r) \iff \text{sat}(\lambda x. P, r)[\text{dom}(R_1)] \odot s[\text{dom}(R_2)] \odot s[\text{dom}(R_1)] \]

def. \( \text{LJD}_{R_1, R_2} (\lambda x. P) \) (1)

\[ \text{set}(\lambda x. P, R) \]

= \( \{ r \mid \forall r \in \text{inh}(R), \text{sat}(\lambda x. P, r) \} \)

def. set(\( \lambda x. \cdot , \cdot \))

= \( \{ r \mid \forall r, s \in \text{inh}(R), \text{sat}(\lambda x. P, r)[\text{dom}(R_1)] \odot s[\text{dom}(R_2)] \odot s[\text{dom}(R_1)] \} \)

(1)

= \( \{ r \mid \forall r \in \text{inh}(R), r[\text{dom}(R_1)] \in \{ s[\text{dom}(R_2)] \mid \forall s \in \text{inh}(R), \text{sat}(\lambda x. P, s) \} \}

\]

Lemma 6

= \( \{ r \mid \forall r \in \text{inh}(R), r[\text{dom}(R_1)] \in \text{set}(\lambda x. P, R)[R_1] \text{ and } r[R_2] \in \text{set}(\lambda x. P, R)[R_2] \}

\]

def. \( \cdot , \cdot \)

= \( \text{set}(\lambda x. P, R)[\text{dom}(R_1)] \cong \text{set}(\lambda x. P, R)[\text{dom}(R_2)] \)

\]

Lemma 2 Given a predicate \( \lambda x. P \) and record types \( R, R' \), it follows that \( \text{LJD}^1_{R, R'} (\lambda x. P) \) implies \( \text{LJD}^2_{R, R'} (\lambda x. P) \).

Proof.

\[ r_1, r_2 \in \text{inh}(R) \]

assumption

\[ s_1, s_2 \in \text{inh}(R') \]

assumption

\[ \text{sat}(\lambda x. P, r_1 \otimes s_1) \]

assumption (1)

\[ \text{sat}(\lambda x. P, r_2 \otimes s_2) \]

assumption (2)

Perform induction on \( \text{LJD}^2_{R, R'} (\lambda x. P) \)

\[ \text{LJD}^1_{R, R'} (\lambda x. P) \]

assumption

\[ \text{Extensionality (1)} \]

\[ \text{Extensionality} \]

def. LJD
Lemma 3 Suppose $R = R_1 \cup R_2$, $R_r \vdash r$ and $LJD_{R_1, R_2} (\lambda x. P)$. Then $DV_{R_1, R_2} (\lambda x. P) r$ implies $r \in \text{set}(\lambda x. P, R)[\text{dom}(R_2)]$.

Proof.

$R = R_1 \cup R_2$

$\exists s \in \text{inh}(R_1). \text{sat}(\lambda x. P, r \otimes s)$

$\implies \exists s \in \text{inh}(R). s[\text{dom}(R_2)] = r$ and $\text{sat}(\lambda x. P, s)$

$\implies r \in \{s[\text{dom}(R_2)] \mid \forall s \in \text{inh}(R). \text{sat}(\lambda x. P, r)\}$

$\implies r \in \text{set}(\lambda x. P, R)[\text{dom}(R_2)]$

Lemma 4 Given a predicate $\lambda x. P$ such that set$(\lambda x. P, R \otimes R')$ is not empty and record $r$ of type $R$, it follows that $DV_{R, R'} (\lambda x. P) r$ implies $DV_{R, R'} (\lambda x. P) r$.

Proof.

$r \in \text{inh}(R')$

set$(\lambda x. P, R \otimes R')$ not empty

(1)

Perform induction on $DV^{t}_{R, R'} (\lambda x. P) r$

$DV^{t}_{R} (\lambda x. P) r$

$x : R \vdash P : D$

$DV^{t}_{R, R'} (\lambda x. P) r$

$\exists s \in \text{inh}(R). \text{sat}(\lambda x. P, s)$

$\exists s \in \text{inh}(R). \text{sat}(\lambda x. P, r \otimes s)$

$DV_{R, R'} (\lambda x. P) r$

$DV^{t}_{R} (\lambda x. P) r$

$DV^{t}_{R, R'} (\lambda x. P) r$
\[ \exists s \in \text{inh}(R). \, \text{sat}(\lambda x. P, r \otimes s) \]

\[ \text{DV}_{R,R'}(\lambda x. P) r \]

\[ \text{DV}_{R,R'}^\top(\lambda x. Q_1) r \]

\[ \text{DV}_{R,R'}^\top(\lambda x. Q_2) r \]

\[ \exists s \in \text{inh}(R). \, \text{sat}(\lambda x. Q_1, r \otimes s) \]

\[ \exists s \in \text{inh}(R). \, \text{sat}(\lambda x. Q_2, r \otimes s) \]

\[ \exists s \in \text{inh}(R). \, \text{sat}(\lambda x. Q_1 \land Q_2, r \otimes s) \]

\[ \text{DV}_{R,R'}^\top(\lambda x. P) r \]

Lemma 14. Suppose \( \Gamma \vdash \text{drop } \ell' \text{ determined by } (\vec{\ell}, v) \text{ from } M : \text{lens of } (\Sigma, R, \lambda x. Q, \lambda y. S, \vec{M}) \text{ and } \langle \vec{M} \rangle = \Sigma / L / S \text{ such that } L \in \Sigma \Leftrightarrow \{ S \} \) and sort \( S = (\text{dom}(R) \cup \ell', P, \mathcal{F}) \). Then \( \langle \text{drop } \ell' \text{ determined by } (\vec{\ell}, v) \text{ from } M \rangle = \Sigma / L' / T \text{ such that } L' \in \Sigma \Leftrightarrow \{ T \} \) and sort \( T = (\text{dom}(R), \text{set}(\lambda x. Q, R, \lambda y. S, \lambda z. T)) \).

Proof.

T-Drop

\[ \mathcal{F} \equiv \mathcal{S} \cup \{ \vec{\ell} \rightarrow \ell' \} \]

\[ \Gamma \vdash M : \text{lens of } (\Sigma, R \cup \{ \ell' : \tau \}, \lambda x. P, \mathcal{F}) \]

\[ \vec{\ell}' = \text{dom}(R) \]

\[ \Gamma \vdash V : A \]

\[ \text{DV}_{R,R'}(\lambda x. P) \{ \ell' = V \} \]

\[ \text{LJD}_{R,R' \langle (\ell', A) \rangle}(\lambda x. P) \]

\[ \text{assumption (1)} \]

\[ \text{assumption (2)} \]

\[ \text{assumption (3)} \]

\[ \text{Lemma 1 (4)} \]

\[ \text{Lemma 3 (5)} \]

\[ \text{define (6)} \]

\[ \text{Lemma 10 (7)} \]

\[ \text{trivial (8)} \]

\[ \text{T-DROP-RL}(3, 8, 1, 7, 4, 5) \]

\[ \text{T-DROP-RL}(3, 8, 1, 7, 4, 5) \]

\( L_d = \text{drop } \ell \text{ determined by } (\vec{\ell}, V) \text{ from } S \) as \( T \in \{ S \} \Leftrightarrow \{ T \} \)

\[ \text{T-DROP-RL}(3, 8, 1, 7, 4, 5) \]

C.4 Translation Sound

Theorem 6. Suppose \( \Gamma \vdash M : \text{lens of } (\Sigma, R, \lambda x. P, \mathcal{F}) \) and \( \langle \vec{M} \rangle = \Sigma / L / T \) then \( L \in \Sigma \Leftrightarrow \{ T \} \) and sort \( T = (\text{dom}(R), \text{set}(\lambda x. P, R, \mathcal{F})) \).

Proof.

perform induction on \( \Gamma \vdash M : \text{lens of } (\Sigma, R, \lambda x. P, \mathcal{F}) \)

\[ \Gamma \vdash \text{lens } M \text{ with } \mathcal{F} : \text{lens of } (\langle \{ T \} \rangle, R, \lambda x. \text{true}, \mathcal{F}) \]

\[ \langle \text{lens } M \text{ with } \mathcal{F} \rangle = \langle \{ T \} \rangle / L / T \]

\[ L \in \{ T \} \Leftrightarrow \{ T \} \]

\[ \text{Lemma 11 (1)} \]

\[ \text{Lemma 11 (1)} \]
Rudi Horn, Simon Fowler, and James Cheney

\[ \begin{align*}
\Gamma \vdash \text{select}_{\lambda x}. Q \text{ from } M : \text{lens of } (\Sigma, R, \lambda x. P \land Q, F) \\
\Gamma \vdash M : \text{lens of } (\Sigma, R, \lambda x. P, F) \\
\langle M \rangle = \Sigma/L/S \\
L \in \Sigma \iff \{S\} \\
\langle \text{select}_{\lambda x}. Q \text{ from } M \rangle = \Sigma/L'/T \\
L' \in \Sigma \iff \{T\} \\
\end{align*} \]

Lemma 12 (2, 3)

\[ \begin{align*}
\Gamma \vdash \text{join } M \text{ with } N \text{ delete_left} : \text{lens of } (\Sigma \cup \Delta, R \cup R', \lambda x. P \land Q, F \cup \emptyset) \\
\Gamma \vdash M : \text{lens of } (\Sigma, R, \lambda x. P, F) \\
\langle M \rangle = \Sigma/L_1/S_1 \\
L_1 \in \Sigma \iff S_1 \\
\langle N \rangle = \Delta/L_2/S_2 \\
L_2 \in \Delta \iff S_2 \\
\langle \text{join } M \text{ with } N \text{ delete_left} \rangle = \Sigma \cup \Delta/L/T \\
L \in \Sigma \cup \Delta \iff \{T\} \\
\end{align*} \]

Lemma 13 (4, 5, 6)

\[ \begin{align*}
\Gamma \vdash \text{drop } \ell' \text{ determined by } (\overrightarrow{t}, v) \text{ from } M : \text{lens of } (\Sigma, R, \lambda x. Q, F) \\
\Gamma \vdash M : \text{lens of } (\Sigma, R', \lambda x. Q, \emptyset) \\
\langle M \rangle = \Sigma/L/T \\
L \in \Sigma \iff \{T\} \\
\langle \text{drop } \ell' \text{ determined by } (\overrightarrow{t}, v) \text{ from } M \rangle = \Sigma/L/T \\
L \in \Sigma \iff \{T\} \\
\end{align*} \]

Lemma 14 (7, 8)