Language-Integrated Query for Temporal Data

(Draft)

Simon Fowler
simon.fowler@glasgow.ac.uk
University of Glasgow
UK

Vashti Galpin
vashti.galpin@ed.ac.uk
University of Edinburgh
UK

James Cheney
jcheney@inf.ed.ac.uk
University of Edinburgh
UK

Abstract

Modern applications often manage time-varying data. Despite decades of research on temporal databases culminating in the addition of temporal data operations into the SQL:2011 standard, these temporal data query and manipulation capabilities are unavailable in most mainstream database management systems, leaving users with the unenviable task of implementing them scratch. In this paper, we extend language-integrated query to support writing temporal queries and updates in a uniform host language, with the language performing the required underlying rewriting to emulate temporal capabilities automatically on any standard relational database. We introduce two core languages, $\lambda_{\text{LINQ}}$ and $\lambda_{\text{VLINQ}}$, for manipulating transaction time and valid time data respectively, and formalise existing implementation strategies by giving provably correct semantics-preserving translations into a non-temporal core language, $\lambda_{\text{LINQ}}$. We show how existing work on query normalisation supports a surprisingly simple implementation strategy for sequenced joins. We implement our approach in the Links programming language, and describe a non-trivial case study based on curating COVID-19 statistics.

1 Introduction

Most interesting programs access or query data stored persistently, often in a database. Relational database management systems (RDBMSs) are the most popular option and provide a standard domain-specific language, SQL, for querying and modifying the data. Ideally, programmers can express the desired queries or updates declaratively in SQL and leave the database to decide how to answer queries or perform updates efficiently and safely (e.g. in the presence of concurrent accesses), but there are many pitfalls arising from interfacing with SQL from a general-purpose language, leading to the well-known impedance mismatch problem [9]. These difficulties range from run-time failures due to the generation of queries as unchecked SQL strings at runtime, to serious security vulnerabilities like SQL injection attacks [29].

Among the most successful approaches to overcome the above challenges, and the approach we build upon in this paper, is language-integrated query, exemplified by Microsoft’s popular LINQ for .NET [22, 33] and in a number of other language designs such as Links [8, 20] and libraries such as Quill [26]. Within this design space we focus on a family of techniques derived from foundational work by Buneman et al. [2] on the nested relational calculus (NRC), a core query language with monadic collection types; work by Wong [34] on rewriting NRC expressions to normal forms that can be translated to SQL, which forms the basis of the approach taken in Links [7, 20] and has been adapted to F# [4].

Many interesting database applications involve data that changes over time. Perhaps inevitably, temporal data management [16] has a long history. Temporal databases provide powerful features for querying and modifying data that changes over time, and are particularly suitable for modeling time-varying phenomena, such as enterprise data or evolving scientific knowledge, and supporting auditing and transparency about how the current state of the data was achieved, for example in financial or scientific settings.

To illustrate how temporal databases work and why they are useful, consider the following toy example: a temporal to-do list. A temporal database can be conceptualised abstractly as a function mapping each possible time instant (e.g. times of day) to a database state [17]. For efficiency in the common case where most of the data is unchanged most of the time, temporal databases are often represented by augmenting each row with an interval timestamp indicating the time period when the row is present. For technical reasons, closed-open intervals $[start, end)$ representing the times $start \leq t < end$ are typically used [31].

In our temporal to-do list, the table at each time instant has fields “task”, a string field, and “done”, a Boolean field. Additional fields “start” and “end” record the endpoints of the time interval during which each row is to be considered part of the database. An end time of “forever” (\(\infty\)) reflects that there is no (currently known) end time and in the absence of other changes, the row is considered present from the start time onwards. For example:

```
<table>
<thead>
<tr>
<th>task</th>
<th>done</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go shopping</td>
<td>true</td>
<td>11:00</td>
<td>(\infty)</td>
</tr>
<tr>
<td>Cook dinner</td>
<td>false</td>
<td>11:00</td>
<td>17:30</td>
</tr>
<tr>
<td>Walk the dog</td>
<td>false</td>
<td>11:00</td>
<td>(\infty)</td>
</tr>
<tr>
<td>Cook dinner</td>
<td>true</td>
<td>17:30</td>
<td>(\infty)</td>
</tr>
<tr>
<td>Watch TV</td>
<td>false</td>
<td>11:00</td>
<td>19:00</td>
</tr>
</tbody>
</table>
```

represents a temporal table where all four tasks were added at 11:00, with “Go shopping” being complete and the others incomplete; at 17:30 “Cook dinner” was marked “done” from then onwards, and at 19:00 “Watch TV” was removed from the table without being completed. Technically, note that...
we take first steps towards reconciling temporal data management in the tradition of TSQL2 with language-integrated query. Temporal data management has the potential to become a "killer app" for language-integrated query, and make temporal data management safer, easier to use and more accessible to non-experts than the current state of the art. As an initial test of this hypothesis we present a more accessible to non-experts than the current state of affairs. We show how existing work on query normalisation can be implemented efficiently in different ways. We propose supporting temporal capabilities by translation to ordinary language-integrated query and hypothesis that this approach can make temporal data management safer, easier to use and more accessible to non-experts than the current state of affairs. As an initial test of this hypothesis we present a high-level design, a working implementation, and detail our experience with a nontrivial case study.

Although both language-integrated query and temporal databases are now well-studied topics, we believe that their combination has never been considered before. Doing so has a number of potential benefits, including making the power of well-studied language designs such as TSQL2 more accessible to non-expert programmers, and providing a high-level abstraction that can be implemented efficiently in different ways. Our interest is particularly motivated by the needs of scientific database development, where data versioning for accountability and research integrity are very important needs that are not well-supported by conventional database systems [3]. Temporal data management has the potential to become a "killer app" for language-integrated query, and this paper takes a first but significant step towards this goal.

The overarching contribution of this paper is the first extension of language-integrated query to support transaction time and valid time temporal data. Concretely, we make three main contributions:

1. Based on $\lambda_{\text{LINQ}}$ (§2), a formalism based on the Nested Relational Calculus (NRC) [28], we introduce typed $\lambda$-calculus to model queries and modifications on transaction time (§3) and valid time (§4) databases. We give semantics-preserving translations to $\lambda_{\text{LINQ}}$ for both.

2. We show how existing work on query normalisation allows a surprisingly straightforward implementation strategy for sequenced joins (§5).

3. We implement our constructs in the Links functional web programming language, and describe a case study based on curating COVID-19 data (§6).

### Figure 1. Syntax of $\lambda_{\text{LINQ}}$

Section 7 discusses related work and Section 8 concludes. Many details and all proofs are relegated to appendices.

## 2 Background: Language-Integrated Query

We begin by introducing a basic $\lambda$-calculus, called $\lambda_{\text{LINQ}}$, to model language-integrated query in non-temporal databases. Our calculus is based heavily on the Nested Relational Calculus [28], with support for database modifications heavily inspired by the calculus of Fehrenbach and Cheney [12]. The calculus uses a type-and-effect system to ensure database accesses can occur in "safe" places, i.e., that we do not attempt to perform a modification operation in the middle of a query. Effects include read (denoting a read from a database) and write (denoting an update to the database).

Types $A, B$ include base types $C$, effect-annotated function types $A \to E B$, unordered collection types $Bag(A)$, record types $\{ t_i : A_i \}$ denoting a record with labels $t_i$ and types $A_i$, and handles Table$(A)$ for tables containing records of type $A$. A record is a base record if it contains only fields of base type. We assume that the base types include at least Bool and the Time type, which denotes (abstract) timestamps. Basic terms include variables $x, c$ (constants), table handles $t$, functions $\lambda x.M$, application $M N$, n-ary operations $\{ M \}$, and conditionals if $L$ then $M$ else $N$. We assume that the set of operations contains the usual unary and binary relation operators, as well as the n-ary operations greatest and least on timestamps which return their largest and smallest arguments respectively. We assume that the set of constants contains timestamps $t$ of type Time, and forever of type Time, which denotes the maximum timestamp. The calculus also includes the empty multiset constructor $\emptyset$; the singleton multiset constructor $\{ M \}$; multiset union $M \uplus N$; and comprehensions for $(x \leftarrow M) N$. We write $\{ M_1, \ldots, M_n \}$ as sugar for $\{ M_1 \} \uplus \ldots \uplus \{ M_n \}$. We also have records $(t_i : M_i)$ and projections $M.t$. Term now retrieves the current time.

We write $let x = N$ in $N$ as the usual syntactic sugar for $(\lambda x.N) M$, and $M; N$ as sugar for $(\lambda x.N)M$ for some fresh $x$. We also define where $M; N$ as sugar for if $M$ then $N$ else $\emptyset$. We also denote unordered collections with a tilde (e.g., $\tilde{M}$), and ordered sequences with an arrow (e.g., $A \rightarrow B$).

<table>
<thead>
<tr>
<th>Types</th>
<th>$A, B$ ::= $C \rightarrow E B \mid Bag(A) \mid (\ell : A) \mid Table(A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base types</td>
<td>$C$ ::= String $\mid$ Int $\mid$ Bool $\mid$ Time</td>
</tr>
<tr>
<td>Effects</td>
<td>$e$ ::= read $\mid$ write</td>
</tr>
<tr>
<td>Effect sets</td>
<td>$E$ ::= $x \mid c \mid t$</td>
</tr>
</tbody>
</table>
| Terms         | $L, M, N$ ::= $x \mid c \mid t \mid \lambda x.M \mid M N \mid \{ \ell \} \mid [M] \mid M \uplus N \mid$  
|               | $(t : M) \mid M.t \mid$ now $\mid$ query $M \mid$ get $M$ $\mid$ insert $M$ values $N$ $\mid$ update $(x \leftarrow L) \mid$ where $M$ set $(t \equiv N)$ $\mid$ delete $(x \leftarrow M) \mid$ where $N$ $\mid$ |

This example interprets the time annotations as transaction time, that is, the times indicate when certain data was in the database; there is another dimension, valid time, and we will discuss both dimensions in greater detail later on.

The problems of querying and updating temporal databases have been well-studied, leading to the landmark language design TSQL2 [30] extending SQL. However, despite decades of effort, only limited elements of TSQL2 were eventually incorporated into the SQL:2011 standard [19] and these features have not yet been widely adopted. Directly implementing temporal queries in SQL is possible, but painful: a TSQL2-style query or update operation may grow by a factor of 10 or more when translated to plain SQL. So these powerful capabilities remain outside the grasp of non-experts. In this paper we take first steps towards reconciling temporal data management in the tradition of TSQL2 with language-integrated query based on query normalisation. We propose supporting temporal capabilities by translation to ordinary language-integrated query and hypothesis that this approach can make temporal data management safer, easier to use and more accessible to non-experts than the current state of affairs. As an initial test of this hypothesis we present a high-level design, a working implementation, and detail our experience with a nontrivial case study.

Although both language-integrated query and temporal databases are now well-studied topics, we believe that their combination has never been considered before. Doing so has a number of potential benefits, including making the power of well-studied language designs such as TSQL2 more accessible to non-expert programmers, and providing a high-level abstraction that can be implemented efficiently in different ways. Our interest is particularly motivated by the needs of scientific database development, where data versioning for accountability and research integrity are very important needs that are not well-supported by conventional database systems [3]. Temporal data management has the potential to become a "killer app" for language-integrated query, and this paper takes a first but significant step towards this goal.

The overarching contribution of this paper is the first extension of language-integrated query to support transaction time and valid time temporal data. Concretely, we make three main contributions:

1. Based on $\lambda_{\text{LINQ}}$ (§2), a formalism based on the Nested Relational Calculus (NRC) [28], we introduce typed $\lambda$-calculus to model queries and modifications on transaction time (§3) and valid time (§4) databases. We give semantics-preserving translations to $\lambda_{\text{LINQ}}$ for both.

2. We show how existing work on query normalisation allows a surprisingly straightforward implementation strategy for sequenced joins (§5).

3. We implement our constructs in the Links functional web programming language, and describe a case study based on curating COVID-19 data (§6).
The get M term retrieves the contents of a table into a bag; insert M values N inserts values N into table M; update (x ⇑ L) where M set (t_i = N_i) updates table L, updating the fields t_i to N_i of each record x satisfying predicate M. The delete (x ⇑ M) where N term removes those records x in table M satisfying predicate N.

2.1 Typing rules

Figure 2 shows the typing rules for \( \lambda_{\text{LINQ}} \). The rules are parameterised by a database schema \( \Sigma \) mapping table names to types of the form \( \text{Bag}(\langle t_i : C_i \rangle) \). Many rules are similar to those of the simply-typed \( \lambda \)-calculus extended with monadic collection operations [2] and a set-based effect system [21], and such standard rules are relegated to Appendix A.

Rule T-QUERY states that a term query M is well-typed if M is of a query type: either a base type, a record type whose fields are query types, or a bag whose elements are query types. The term M must also only have read effects.

Rule T-GET states that get M has type Bag(A) if M has type Table(A), and produces the read effect. Rule T-TABLE states that a table reference follows the type of the table in the schema. T-INSERT types a database insert insert M values N, requiring M to be a table reference of type Table(A), and the inserted values N to be a bag of type Bag(A). T-UPDATE ensures the predicate and update terms are typable under an environment extended with the row type, and ensures that all updated values match the type expected by the row. Rule T-DELETE is similar. All subterms used as predicates or used to calculate updated terms must be pure (that is, side-effect free), and all modifications have the write effect.

2.2 Semantics

Figure 3 shows the syntax and typing rules of values, and the big-step semantics of \( \lambda_{\text{LINQ}} \). Most rules are standard, and presented in Appendix A. Values V, W include functions, constants, tables, fully-evaluated records, and fully-evaluated bags \( \{ V \} \). Unlike the unary bag constructor \( \{ M \} \), fully-evaluated bags contain an unordered collection of values. All values are pure. We write \( \oplus \) for record extension, e.g., \( (t_1 = M) \oplus (t_2 = N) = (t_1 = M, t_2 = N) \).

Since evaluation is effectful (as database operations can update the database), the evaluation judgement has the shape \( M \downarrow_{\Delta} (V, \Delta') \); this can be read "term M with current database \( \Delta \) at time \( t \) evaluates to value V with updated database \( \Delta' \)." A database is a mapping from table names to bags of base records. To avoid additional complexity, we assume evaluation is atomic and does not update the time; one could straightforwardly update the semantics with a tick operation without affecting any results.

We use two further evaluation relations for terms which do not write to the database: \( M \downarrow^* V \) states that a pure term M (i.e., a term typable under an empty effect set) evaluates to V. Similarly, \( M \downarrow^* V \) states that a term M which only performs the read effect evaluates to V. We omit the definitions, which are similar to the evaluation relation but do not propagate database changes (since no changes can occur).

Rule E-Now returns the current timestamp. Rule E-Query evaluates the body of a query using the read-only evaluation relation. Rule E-Get evaluates its subject to a table reference, and then returns the contents of a table. Rule E-Insert does similar, evaluating the values to insert, and then appending them to the contents of the table. Rule E-Update iterates over a table, updating a record if the predicate matches, and leaving it unmodified if not. Finally, E-Delete deletes those rows satisfying the deletion predicate.

\( \lambda_{\text{LINQ}} \) enjoys a standard type soundness property.

**Proposition 2.1 (Type soundness).** If \( \vdash M : A \) then there exists some \( V \) and \( \Delta' \) such that \( M \downarrow_{\Delta'} (V, \Delta') \) and \( \vdash V : A \).

More importantly, the type-and-effect system ensures that query and update expressions in \( \lambda_{\text{LINQ}} \) can be translated to SQL equivalent, even in the presence of higher-order functions and nested query results [4, 5, 7, 20]. This alternative implementation is equivalent to the semantics given here but usually much more efficient since the database query optimiser can take advantage of any available integrity constraints or statistics about the data.

3 Transaction Time

The first dimension of time we investigate is transaction time [32], which records how the state of the database changes over time. The key idea behind transaction time databases is that update operations are non-destructive, so we can always view a database as it stood at a particular point in time.

Let us illustrate with the to-do list example from the introduction. The original table is on the left. The table after making some changes is shown on the right.

![Table](Image)

However, since updates and deletions in \( \lambda_{\text{LINQ}} \) are destructive, we have lost the original data. Instead, let us see how this could be handled by a transaction-time database:

![Table](Image)

There are several methods by which we can maintain the temporal information in the database: for example we could maintain a tracking log which records each entry, or we could use various temporal partitioning strategies [31]. In this paper, we use a period-stamped representation, where each record in the database is augmented with fields delimiting the interval when the record was present in the database.
3.1 Calculus

$\lambda_{\text{LINQ}}$ extends $\lambda_{\text{LINQ}}$ with native support for transaction time operations; instead of performing destructive updates, we adjust the end timestamp of affected rows and, if necessary, insert updated rows. $\lambda_{\text{LINQ}}$ database entries are therefore of the form $V^i_1 \{ V_2, V_3 \}$, where $V_1$ is the record data and $V_2$ and $V_3$ are the start and end timestamps.

Figure 4 shows the syntax, typing rules, and semantics of $\lambda_{\text{LINQ}}$; for brevity, we show the main differences to $\lambda_{\text{LINQ}}$. Period-stamped transaction-time database rows are represented as triples data$(\text{start, end})$ with type TransactionTime(A), where data has type A (the type of each record), and both start and end have type Time. A row is currently present in the database if its end value is forever. We introduce three accessors: data V extracts the data record from a transaction-time row; start V extracts the start time; and end V extracts the end time. The get construct has an updated type to show that it returns a bag of TransactionTime(A) values, rather than the records directly. The typing rules for the other constructors remain the same as in $\lambda_{\text{LINQ}}$.

The accessor rules ET-Data, ET-Start, and ET-End project the expected component of the transaction-time row. Rule ET-Insert period-stamps each record to begin at the current time, and sets the end time to be forever. Rule ET-Delete
Additional Syntax for $\lambda_{TLINQ}$

Types  
\[ A, B ::= \cdots \mid \text{TransactionTime}(A) \]

Terms  
\[ L, M, N ::= \cdots \mid \text{data} M \mid \text{start} M \mid \text{end} M \]

Modified Typing Rules for $\lambda_{TLINQ}$

T-Row
\[ \Gamma \vdash V: A \rightarrow \emptyset \]
\[ \Gamma \vdash V: \text{TransactionTime} \rightarrow \emptyset \]
\[ \Gamma \vdash V[1, V_2]: \text{TransactionTime}(A) \rightarrow \emptyset \]

T-Data
\[ \Gamma \vdash M: \text{TransactionTime}(A) \rightarrow E \]
\[ \Gamma \vdash \text{data} M: A \rightarrow E \]
\[ \Gamma \vdash \text{M} \rightarrow M: \text{TransactionTime}(A) \rightarrow E \]
\[ \Gamma \vdash \text{start} M: \text{TransactionTime}(A) \rightarrow E \]
\[ \Gamma \vdash \text{end} M: \text{TransactionTime}(A) \rightarrow E \]

T-Get
\[ \Gamma \vdash \text{get} M: \text{Table}(A) \rightarrow E \]
\[ \Gamma \vdash \text{get} M: \text{Table}(A) \rightarrow (\text{read} \cup E) \]

Semantics for $\lambda_{TLINQ}$ database operations

ET-Data
\[ M \uparrow_{A, (V, \Lambda')} \]
\[ \text{data} M \uparrow_{A, (V, \Lambda')} \]
\[ \text{start} M \uparrow_{A, (V, \Lambda')} \]
\[ \text{end} M \uparrow_{A, (V, \Lambda')} \]

ET-Delete
\[ \Delta_2 = \Delta_1 \{ [ t \mapsto \text{del}(v) \mid v \in \Delta_1(t)] \} \]
\[ \text{del}([\text{data}]) \]
\[ \text{del}([\text{start}, \text{end}]) \]

ET-Update
\[ \Delta_2 = \Delta_1 \{ [ t \mapsto \text{upd}(v) \mid v \in \Delta_1(t)] \} \]
\[ \text{upd}([\text{data}]) \]
\[ \text{upd}([\text{start}, \text{end}]) \]

ET-Insert
\[ M \uparrow_{A, (V, \Lambda')} \]
\[ \text{vs} = \{ [ v ] \}_{\forall v \in V} \]
\[ \Delta_2 = \Delta_1 \{ [ t \mapsto \text{vs} ] \} \]

Figure 4. Syntax, typing rules, and semantics of $\lambda_{TLINQ}$. Auxiliary functions $\text{del}$ and $\text{upd}$ are local to the respective rules.

3.2 Translation

We can implement the native transaction-time semantics for $\lambda_{TLINQ}$ database operations by translation to $\lambda_{LINQ}$. Our translation adapts the SQL implementations of temporal operations by Snodgrass [31] to a language-integrated query setting. We prove correctness relative to the semantics. $\lambda_{TLINQ}$ represents period-stamped data as nested records, whereas $\lambda_{LINQ}$ requires table types to be flat. Consequently, the translations require knowledge of the types of each record. We therefore annotate each $\lambda_{TLINQ}$ database term with the type of table on which it operates (this can be achieved through a standard type-directed translation pass).

The (omitted) translation of $\lambda_{TLINQ}$ types into $\lambda_{LINQ}$ types is straightforward, save for TranslationTime$(A)$ which is translated into a record $([\text{data}[]], \text{start}: \text{Time}, \text{end}: \text{Time})$. The same is true for the basic $\lambda$-calculus terms. Timestamped rows $V_{\text{start}}V_{\text{end}}$ are translated to fit the above record type; specifically, $(\text{data} = [\text{data}[]], \text{start} = [\text{V}_{\text{start}}], \text{end} = [\text{V}_{\text{end}}])$.

We define the flattening of a $\lambda_{TLINQ}$ row and database as:

\[ \downarrow (t_i = V_i[V_{W_i}]) = (t_i = V_i) \cup (\text{start} = W_t, \text{end} = W_e) \]
\[ \downarrow \Delta = [ t \mapsto [D]_t \mid t \mapsto [D]_t \in \Delta ] \]

Figure 5 shows the translation of $\lambda_{TLINQ}$ terms into $\lambda_{LINQ}$. The translation makes use of three auxiliary definitions.
Auxiliary Definitions

\[ \eta(x, \bar{t}) \triangleq (\ell_t = x, \ell_t)_t \]
\[ \text{restrict}(x, \bar{t}, M) \triangleq (\lambda x. M) \eta(\bar{t}, x) \]
\[ \text{isCurrent}(M) \triangleq M. \text{end} = \text{forever} \]

Translation on database terms

\[
\begin{align*}
[\text{data } M] & = [M]. \text{data} \\
[\text{start } M] & = [M]. \text{start} \\
[\text{end } M] & = [M]. \text{end} \\
[\text{get}(\ell_t, A_\ell), M] & = \\
\text{query} \\
& \{ (x \leftarrow \text{get } [M]) | (\text{data} = \eta(x, \bar{t}), \text{start} = x. \text{start}, \text{end} = x. \text{end}) \} \\
\end{align*}
\]

\[
\begin{align*}
\text{insert}(\ell_t, A_\ell), [M] \text{ values } N] & = \\
\text{let rows} = \\
& \{ \eta(x, \bar{t}) \oplus (\text{start} = \text{now}, \text{end} = \text{forever}) \} \\
\text{in} \\
\text{insert } [M] \text{ values } N \\
\end{align*}
\]

\[
\begin{align*}
\text{delete}(\ell_t, A_\ell), (x \leftarrow M) \text{ where } N] = \\
\text{update} (x \leftarrow [M]) \\
& \{ \text{restrict}(x, \bar{t}, [N]) \land \text{isCurrent}(x) \} \\
\text{set } (\text{end} = \text{now}) \\
\end{align*}
\]

\[
\begin{align*}
\text{update}(\ell_t, A_\ell); (x \leftarrow L) \text{ where } M \text{ set } (\ell = N_j)_{j \in J} = \\
\text{let } tbl = [L] \in \\
\text{let affected} = \\
\text{query} \\
& \{ (x \leftarrow \text{get } tbl) \\
& \{ \text{restrict}(x, \bar{t}; [M]) \land \text{isCurrent}(x) \} \\
& \ell_t \leftarrow \ell_t \oplus 1, \ell_t \oplus 1 \oplus (\ell_j = \text{restrict}(x, \bar{t}; [N_j])_{j \in J} \oplus (\text{start} = \text{now}, \text{end} = \text{forever}) \} \\
\text{in} \\
\text{update} (x \leftarrow tbl) \\
& \{ \text{restrict}(x, \bar{t}; [M]) \land \text{isCurrent}(x) \} \\
\text{set } (\text{end} = \text{now}) \\
\text{insert } tbl \text{ values affected} \\
\end{align*}
\]

Figure 5. Translation from \( \lambda_{\text{LINQ}} \) into \( \lambda_{\text{T LINQ}} \)

3.3 Metatheory

We restrict our attention to well formed rows and databases, where the start timestamp is less than the end timestamp.

Definition 3.1 (Well formed rows and databases). A database \( \Delta \) is well formed, written \( \text{wf}(\Delta) \), if every timestamped row \( V_{\text{data}}[V_{\text{start}}, V_{\text{end}}] \) it satisfies \( V_{\text{start}} < V_{\text{end}} \).

Definition 3.2 (Maximum timestamp). The maximum timestamp of a collection of records \( \bar{D} = \{V_{\text{data}}[V_{\text{start}}, V_{\text{end}}]\}_i \), written \( \text{max}(\bar{D}) \), is the greatest \( V_{\text{end}} \), such that \( V_{\text{end}} \neq \text{forever} \). The maximum timestamp of a database \( \Delta \), written \( \text{max}(\Delta) \), is the maximum timestamp of all its constituent tables.

Again, \( \lambda_{\text{T LINQ}} \) enjoys type soundness.

Proposition 3.1 (Type soundness (\( \lambda_{\text{T LINQ}} \))). If \( \vdash M : A \rightarrow E \), then given a \( \text{wf}(\Delta) \) and \( i \) such that \( \text{max}(\Delta) \leq i \), then there exists some \( V \) and well formed \( \Delta' \) such that \( M \upharpoonright_{\Delta'}(V, \Delta') \).

We can now show that the translation is correct:

Theorem 3.1. If \( \vdash M : A \rightarrow E \) and \( M \upharpoonright_{\Delta'}(V, \Delta') \) where \( \text{wf}(\Delta) \) and \( \text{max}(\Delta) \leq i \), then \( [M] \upharpoonright_{\Delta'}(\Delta', \Delta') \).

4 Valid Time

The other dimension of time we will look at is valid time, which tracks when something is true in the domain being modelled. Each timestamp therefore defines the period of validity (PV) of each record.

Unlike in a transaction time database, the database does not necessarily grow monotonically since we can apply destructive updates and deletions. Furthermore, whereas in a transaction time database timestamps can only refer to the past (or forever), in a valid time database we may state that a row is valid until some specific point in the future (for example, the end of a fixed-term employment contract). A further difference from transaction time databases is that users can modify timestamps directly, and can also apply updates and deletions over a period time. Let us illustrate with the ‘employees’ table of an HR database:

<table>
<thead>
<tr>
<th>name</th>
<th>position</th>
<th>salary</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Lecturer</td>
<td>40000</td>
<td>2010</td>
<td>2018</td>
</tr>
<tr>
<td>Alice</td>
<td>Senior Lecturer</td>
<td>50000</td>
<td>2018</td>
<td>2023</td>
</tr>
<tr>
<td>Bob</td>
<td>PhD Student</td>
<td>150000</td>
<td>2019</td>
<td>2023</td>
</tr>
<tr>
<td>Charles</td>
<td>PhD Student</td>
<td>150000</td>
<td>2018</td>
<td>2022</td>
</tr>
</tbody>
</table>

The first modification is to hire Dolores as a professor, on an open-ended contract. As this is an insertion operation on the database at the current moment in time, it is known as a current insertion. We can write the following query:

\[ \text{insert employees values} \]

\( \text{(name = "Dolores", position = "Professor", salary = 70000)} \)
Syntax

Types $\mathcal{A}, \mathcal{B} ::= \text{ValidTime}(\mathcal{A})$

Terms $L, M, N ::= \cdots \mid [M;N] \mid \text{start } M \mid \text{end } M \mid \text{insert sequenced } M \text{ values } N$

update sequenced $(x \equiv L)$ between $M_1$ and $M_2$ where $M_2$ set $(\ell = N)$
update nonsequenced $(x \equiv L)$ where $M_2$ set $(\ell = N)$ from $N_1'$ to $N_2'$
delete sequenced $(x \equiv L)$ between $N_1$ and $N_2$ where $M$
delete nonsequenced $(x \equiv L)$ where $N$

Values $V, W ::= \cdots \mid V_i^{[W]}$

Typing rules

TV-GET

$\Gamma \vdash M; \text{Table}(A)! E$
$\Gamma \vdash $get $M; \text{Bag(ValidTime(A))}! \{\text{read}\} \cup E$
$\Gamma \vdash $update sequenced $(x \equiv L)$ between $M_1$ and $M_2$ where $M_1$ set $(\ell = N)$(i) \{\text{write}\} \cup E$
$\Gamma \vdash $update nonsequenced $(x \equiv L)$ where $M$ set $(\ell = N)$ valid from $N_1'$ to $N_2'$(i) \{\text{write}\} \cup E$

TV-SEQINSERT

$\Gamma \vdash M; \text{Table}(A)! E \quad \Gamma \vdash N; \text{Bag(ValidTime(A))}! \emptyset$
$\Gamma \vdash $insert sequenced $M$ values $N_i$(i) \{\text{write}\} \cup E

Figure 6. Syntax and typing rules for $\lambda_{\text{VLINQ}}$ (selected)

Next, we want to record that Alice has resigned. We can write the following current deletion query:

```
delete (x \equiv \text{employees}) \text{ where } x.\text{name} = \text{"Alice"}
```

The resulting table state shows that Dolores is a Professor from the current time onwards, and that the ‘end’ field of Alice’s current row is updated to the current year:

<table>
<thead>
<tr>
<th>name</th>
<th>position</th>
<th>salary</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Lecturer</td>
<td>40000</td>
<td>2010</td>
<td>2018</td>
</tr>
<tr>
<td>Alice</td>
<td>Senior Lectures</td>
<td>50000</td>
<td>2016</td>
<td>2022</td>
</tr>
<tr>
<td>Dolores</td>
<td>Professor</td>
<td>70000</td>
<td>2022</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

A powerful feature of valid-time databases is the ability to perform sequenced modifications, which apply an update or deletion over a particular period of applicability (PA). In fact, current modifications are a special case of sequenced modifications applied from now until forever. Suppose that Dolores has agreed to act as Head of School between 2023 and 2028. We can record this using a sequenced update query:

```
update sequenced (x \equiv \text{employees}) between 2023 and 2025 where (x.\text{name} = \text{"Dolores"}) set (position = \text{"Head of School"})
```

with the resulting table being:

<table>
<thead>
<tr>
<th>name</th>
<th>position</th>
<th>salary</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dolores</td>
<td>Professor</td>
<td>70000</td>
<td>2022</td>
<td>2023</td>
</tr>
<tr>
<td>Dolores</td>
<td>Head of School</td>
<td>70000</td>
<td>2023</td>
<td>2028</td>
</tr>
<tr>
<td>Dolores</td>
<td>Professor</td>
<td>70000</td>
<td>2028</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Since the period of applicability of the sequenced update is entirely contained within the period of validity of Dolores’s row, we end up with three rows: the unchanged record before and after the PA, and the updated record during the PA. We also allow a sequenced deletion, and a sequenced insertion, where each record’s period of validity is given explicitly.

Additionally, suppose that all PhD students are to be given a 1-year extension due to the disruption caused by the pandemic; in this case we want to change the period of validity directly. This is known as a nonsequenced update. We cannot express this modification using either current or sequenced modifications since we must calculate the each row’s new end date from its previous end date. We can write the modification as follows, noting that we can both read from, and write to, the period of validity directly:

```
update nonsequenced (x \equiv \text{employees}) where ((x.\text{data}).\text{position} = \text{"PhD student"}) set (\text{valid from} (x.\text{start}) \to (\text{end} + 1))
```

The resulting table shows that the ‘end’ field of Bob’s and Charles’ records are updated to 2024 and 2023 respectively:

<table>
<thead>
<tr>
<th>name</th>
<th>position</th>
<th>salary</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>PhD Student</td>
<td>15000</td>
<td>2019</td>
<td>2024</td>
</tr>
<tr>
<td>Charles</td>
<td>PhD Student</td>
<td>15000</td>
<td>2018</td>
<td>2023</td>
</tr>
</tbody>
</table>

4.1 Calculus

The $\lambda_{\text{VLINQ}}$ calculus gives a direct semantics to valid time operations. Like $\lambda_{\text{LINQ}}$, $\lambda_{\text{VLINQ}}$ has a native notion of a period-stamped database row, with accessors for the data and each timestamp; the typing rules, reduction rules, and translations are straightforward adaptations of those in $\lambda_{\text{LINQ}}$.

Figure 6 shows how the syntax and typing rules for $\lambda_{\text{VLINQ}}$ differ from those of $\lambda_{\text{LINQ}}$. Unlike in $\lambda_{\text{LINQ}}$, we can use the term $M_i^{[M_0, M_1]}$ to construct a valid-time row. Ordinary insert,
Fig. 7. Reduction rules for $\lambda_{V\text{LINQ}}$ (selected)

delete and update operations may be applied to valid time tables (the straightforward rules are omitted). Sequenced insertions are described by the term \textit{insert sequenced} $M$ \textit{values} $N$ where TV-SEQINSERT ensures that $N$ is a bag of timestamped records. Sequenced updates are described by:

\textbf{update sequenced} $(x \leftarrow L)$ between $M_1$ and $M_2$ where $M_1$ set $(\ell = N)$

Terms $M_1$ and $M_2$ must be of type Time, referring to the period of applicability of the sequenced update. Nonsequenced updates are described by the term:

\textbf{update nonsequenced} $(x \leftarrow L)$ where $M$ set $(\ell = N)$ valid from $N_1'$ to $N_2'$

with TV-NONSEQUPDATE stating that the database row (including period information) is bound as $x$ in the predicate $M$, update terms $N_1$, and new time periods $N_1'$ and $N_2'$. Finally, the term:

\textbf{delete sequenced} $(x \leftarrow L)$ between $M_1$ and $M_2$ where $N$

describes a sequenced deletion which removes the portion of each record satisfying $N$ between times $M_1$ and $M_2$.

Since current insertions, updates, and deletions are special cases of sequenced operations, we need not consider them explicitly; for completeness, direct semantics can be found in Appendix C. Instead, we show macro translations to the sequenced constructs. Current insertions can be implemented by deseguring to sequenced insertions, annotating each row with $[\text{now, forever}]$:

\begin{align*}
\textit{insert} M \textit{values} N \leadsto \textit{let} \textit{rows} \textit{for} (x \leftarrow N) [x[\text{now, forever}] \in \textit{insert sequenced} M \textit{values} \textit{rows}] \end{align*}

Current updates and deletions can be implemented as sequenced updates and deletions where the period of applicability spans from \textit{now} until \textit{forever}:

\begin{align*}
\textit{update} (x \leftarrow L) \textit{where} M \textit{set} (\ell = N_1) \leadsto \\
\textit{update sequenced} (x \leftarrow L) \textit{between now and forever} \textit{where} M \textit{set} (\ell = N_1) \in \\
\textit{delete} (x \leftarrow M) \textit{where} N \leadsto \\
\textit{delete sequenced} (x \leftarrow M) \textit{between now and forever where} N
\end{align*}

Fig. 7 shows selected reduction rules for sequenced operations. We show the rules for sequenced inserts and updates; the rules for other cases employ similar ideas and are included in Appendix A. Nonsequenced updates and deletes are similar to their analogues in $\lambda_{\text{LINQ}}$ but allow access to, and modification of, row timestamps. For sequenced insertions, EV-SEQINSERT checks that the period of validity for each row is correct (i.e., that the \textit{start} field is less than the \textit{end} field) and appends the provided bag to the table. Sequenced updates and deletes must account for the various ways that the period of applicability can overlap the period of validity. There are five main cases, corresponding to the five ways two closed-open intervals can overlap (or fail to do so). Appendix B summarizes the five cases and describes how deletes and updates are handled in each case.
we discuss the translations for sequenced inserts and both a row, the startRows and endRows functions calculate the records which must be inserted before and after the period of applicability. To translate a sequenced update, we calculate the rows to insert, perform an update to set the new values and set the new period of applicability to the overlap between the PA and PV using the greatest and least functions, and finally materialise the insertions. Sequenced deletions (shown in Appendix A) are similar but delete the rows that overlap the PA instead of updating them.

4.3 Metatheory

Evaluation preserves typing and well-formedness.

Proposition 4.1 (Preservation (λVLINQ)). If M A!E and M ⊩ ( V, Δ) for some wf(Δ), then M ⊩ V: A!∅ and wf(Δ').

Unlike λLINQ and λTLINQ, evaluation in λVLINQ is partial in order to reflect the need for dynamic checks that start times precede end times. In practice, our implementation evaluates temporal updates as single transactions and raises an exception (aborting the transaction) when a well-formedness check fails, but our formalisation assumes updates preserve well-formedness in order to avoid clutter.

Our translation from λVLINQ into λTLINQ satisfies the following correctness property:

Theorem 4.1. If M A!E and M ⊩ ( V, Δ) for some wf(Δ), then M ⊩ ( V, Δ')

5 Sequenced Joins

Queries that join multiple tables are straightforward to encode using language integrated query. Keeping with our employee database, say we wish to separate out the salary into a separate table. The non-temporal employee database might look as follows:

<table>
<thead>
<tr>
<th>name</th>
<th>position</th>
<th>band</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Senior Lecturer</td>
<td>A08</td>
<td>40000</td>
</tr>
<tr>
<td>Bob</td>
<td>PhD Student</td>
<td>B01</td>
<td>50000</td>
</tr>
<tr>
<td>Charles</td>
<td>PhD Student</td>
<td>B01</td>
<td>70000</td>
</tr>
<tr>
<td>Dolores</td>
<td>Professor</td>
<td>A10</td>
<td>15000</td>
</tr>
</tbody>
</table>

We can get the salary for each employee as follows:

query for (e ← get employees) 
for (s ← get salaries) where (e.band = s.band) 
\{ name = e.name, salary = s.salary \}

Joining a temporal table with a non-temporal table is also easily expressible. Consider a version of our previous temporal employees table from just after when Dolores joined:

<table>
<thead>
<tr>
<th>name</th>
<th>position</th>
<th>band</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Lecturer</td>
<td>A08</td>
<td>2010</td>
<td>2018</td>
</tr>
<tr>
<td>Alice</td>
<td>Senior Lecturer</td>
<td>A09</td>
<td>2018</td>
<td>∞</td>
</tr>
<tr>
<td>Bob</td>
<td>PhD Student</td>
<td>B01</td>
<td>2019</td>
<td>2023</td>
</tr>
<tr>
<td>Charles</td>
<td>PhD Student</td>
<td>B01</td>
<td>2018</td>
<td>2022</td>
</tr>
<tr>
<td>Dolores</td>
<td>Professor</td>
<td>A10</td>
<td>2022</td>
<td>∞</td>
</tr>
</tbody>
</table>

We can join this table with the non-temporal salaries table as follows; for clarity, we denote valid-time get as getv:
Things get more interesting when both tables are temporal. Salaries are not static over time; bands go up with inflation, so even for a given employee, their salary will change over time. Suppose we now have two temporal tables:

\[ \text{Employees: } (\text{name, band, salary, start, end}) \]
\[ \text{Salaries: } (\text{name, band, salary}) \]

Consider the above table along with a temporal salaries table showing a pay increase in 2015:

- Band A08 was at £38000 from 2015 to 2016.
- Band A09 was at £40000 from 2010 to 2015.
- Band A08 was at £50000 from 2015 to 2016.

What does it mean to join two temporal tables? In essence, we want to record all configurations of a particular joined record, creating new records with shorter periods of validity whenever data from either underlying table changes. Concretely, joining the above two temporal tables would give:

- Alice was on band A08 for three different periods:
  - The first time when Alice was on band A08, her salary was £38000.
  - The second time when band A08 increased to £40000.
  - The third when Alice was promoted to band A09.

Such joins are called sequenced because they (conceptually) evaluate the join on the whole sequence of states encoded by each table. Manually writing the sequenced joins in SQL is error-prone. We instead introduce a construct, `join`, which allows us to write the following:

\[
\text{query for } (e \leftarrow \text{get, employees}) \text{ and } (s \leftarrow \text{get, salaries}) \\
\text{where } ((\text{data } e) . \text{band} = (\text{data } s) . \text{band}) \\
\{ (\text{name} = e . \text{name}, \text{salary} = s . \text{salary}) \text{[start e, end e]} \}
\]

Note that we do not need to calculate the period of validity for each resulting row; this is computed automatically.

Figure 9 shows how sequenced joins can be implemented; we show the constructs for valid time, but the same technique can be used for transaction time. The typing rule requires that the result of a `join` query is flat (nested sequenced queries are conceptually nontrivial).

As mentioned earlier, queries can be rewritten to normal forms for conversion to SQL, as shown in Figure 9. The structure of these normal forms allows sequenced joins to be implemented through a simple rewrite: the `greatest` and `least` functions are used to calculate the intersections of the periods of validity for each combination of records from each generator, with the modified predicate ensuring that the periods of overlap make sense. The calculated overlapping periods of validity are then returned in the resulting row.

6 Implementation and Case Study

The Links programming language [8] is a statically-typed functional web programming language which allows client, server, and database code to be written in a uniform language. We have extended Links with support for the constructs described in Sections 3 and 4, as well as support for temporal joins as described in Section 5. In this section, we describe a case study based on curating COVID-19 data.

Our translations from $\lambda_{\text{UNIQ}}$ into $\lambda_{\text{UNIQ}}$ are trivially realisable in SQL. Queries can be compiled using known techniques (e.g., [7]). The `startsWith` and `endsWith` functions can be compiled using an SQL `WITH` statement, and there is a direct correspondence between $\lambda_{\text{UNIQ}}$ modification operations and their SQL equivalents. Each translated temporal modification is executed as an SQL transaction, with primary key and referential integrity constraint checking deferred until the end of the transaction.

Case study. We have used the temporal features of Links in two prototypes based on curated scientific databases: cura-

...

We concentrate on the first prototype. In 2020, the Scottish Government began releasing various data about the COVID-19 pandemic [23]. This included weekly data of fatalities in Scotland in various categories (such as ‘Sex’). Each weekly release was a CSV file, with a row for each subcategory (‘Sex’ has the subcategories ‘Male’ and ‘Female’, for example) and a column for each week for which data was available. Each release included an additional week column with the latest data (see Figure 10). Importantly, each release could include revisions to data for previous weeks.

Information about the changes to the data over time is often desired to understand its provenance and assess its trustworthiness [3]. From a provenance point of view, this data is interesting because a column for an earlier week may contain updated data. We developed a web application for the querying of the data (‘How do the Male and Female subcategories compare in terms of the change in fatalities from last week to this week?’) as well as querying the changes in the data (‘How do the Male and Female subcategories compare in terms of number of updates to existing values?’).

Considering the non-temporal data, an entry in a database table would be a row consisting of the key fields subcat and weekdate and a value field giving the corresponding count. In the case of the temporal data, the key fields are insufficient to uniquely identify the value of the count because it may have different values over time. Thus the time validity fields are necessary to provide a key for the value.

The prototype uses a valid time table for fatality data to capture the notion that a count value, either brand new or an update, becomes valid as soon as the CSV is uploaded into the interface (this can be a different time from when the new value is accepted and written to the database). In Links it is possible to specify the names of the period stamping fields, which have the built-in type dateTime. This table is defined using the following Links code; we have omitted some details in the code snippets for brevity.

```links
var covid_data =
  table "covid_data"
  with (subcat: Int, weekdate: String, count: Int)
  using valid_time(valid_from, valid_to)
  from database "covid_curation";
```

The prototype’s upload workflow is as follows: the user uploads a new CSV file, and the count values for the new week are added to the database. For counts that pertain to earlier weeks and that now have different values, the user is shown these counts and can accept them, reject them or move them to a pending list for a later decision. In terms of implementation, brand new count values are added to the table with a sequenced insert, using the upload time as the start time. The Links code for this and other examples can be found in Appendix E. The process is more complex for updated count values, because the interface shows the user previous value, to support decision making. This requires a conditional join over the current state of the covid_data table and the count values from the CSV file. If a modification is accepted, it is added using a sequenced update. Figure 10 illustrates how the table changes as a result of a single update.

The prototype also provides functionality to query data, both as current data, and as data with information about changes. The current data is obtained using a current query. The result is a list of weeks and counts grouped by subcategory. This is repeated for each category.

```links
fun getCurrentData (category) {
  query nested {
    for (x <-- subcategory)
      where (x.cat == category)
      [(subcat_name = x.subcat_name, cat = x.cat,
        results =
          for (y <-- vtCurrent(covid_data))
            for (z <-- week)
              where (y.subcat == x.subcat &&
                y.weekdate == z.weekdate &&
                z.all_zero == false)
                [(count = y.count, weekdate = y.weekdate)])
  }
}
```

Instead of an explicit get construct, Links uses the ‘double arrow’ comprehension `--` to represent a nontemporal database query, with `--t-` and `--v-` supporting transaction time and valid time queries respectively. The ‘single arrow’ comprehension `<-` denotes a list comprehension. Finally, `vtCurrent` is a standard library function which performs a valid time query to obtain the values valid at the current time.

For update provenance queries of individual counts, a self join is computed over the subcategory and week fields of the valid time table to provide a nested result table where each count is associated with a list of count values and their associated start and end time information. This is a nonsequenced query because the time period information is explicitly added to the result table. The user can specify the subcategory and week they are interested in, and obtain details of modifications. The interface also supports update provenance by day and by category. This is illustrated in Figure 11.

7 Related and Future Work

Most of the focus of effort on language-integrated query has been (perhaps unsurprisingly) on queries rather than updates, beginning with the foundational work on nested relational calculus by Buneman et al. [2] and on rewriting queries for translation to SQL by Wong [34]. Lindley and Cheney [20] presented a calculus including both query and update capabilities and our type and effect system for tracking database read and write access is loosely based on
Table 1: Example of data uploads, sequenced insertion and sequenced update

<table>
<thead>
<tr>
<th>Week</th>
<th>Subcat</th>
<th>Count</th>
<th>Valid-from</th>
<th>Valid-to</th>
</tr>
</thead>
<tbody>
<tr>
<td>30-Mar-2020</td>
<td>Female</td>
<td>126</td>
<td>&lt;first CSV upload&gt;</td>
<td>&lt;later CSV upload&gt;</td>
</tr>
<tr>
<td>30-Mar-2020</td>
<td>Male</td>
<td>156</td>
<td>&lt;first CSV upload&gt;</td>
<td>&lt;forever&gt;</td>
</tr>
</tbody>
</table>

Updates from the CSV of 13-Apr-2020

New data from the CSV of 30-Mar-2020

Figure 10. Example of data uploads, sequenced insertion and sequenced update

Figure 11. Interface screenshot: history of a count

The ubiquity and importance of time in applications of databases was appreciated from an early stage [6] and led to a significant community effort to standardize temporal extensions to SQL based on the TSQL2 language design in the 1990s and early 2000s [30]. This effort ultimately resulted in standardisation of a relatively limited subset of the original proposal in SQL:2011 [19]. Since then temporal database research has progressed steadily, including recent contributions showing how to implement temporal data management as a layer on top of a standard RDBMS [10], and establishing connections between temporal querying and data provenance and annotation models [11].

Snodgrass [31] describes how to implement TSQL2-style updates and queries by translation to SQL, but we are not aware of previous detailed formal proofs of correctness of translations for transaction time and valid time updates. Although timestamping rows with time intervals is among the most popular ways to represent temporal databases as flat relational tables, it is not the only possibility. Jensen et al. [17] proposed a bitemporal conceptual data model that captures the abstract meaning of a temporal table and used it to compare different representation strategies.

There can be multiple representations of the same abstract temporal data, leading to consideration of the problem of coalescing or normalizing the intervals to save space and avoid ambiguity. Nonsequenced updates can be used to perform modifications that have different effects on representations of the same conceptual table. We have not considered coalescing or other common issues such as how to handle operations such as deduplication, grouping and aggregation (including emptiness testing), or integrity constraints in a temporal setting. Some of these issues appear orthogonal to the high-level language design and could be incorporated “under the hood” into the implementation or even performed directly on the database.

One important future application is to retrofit temporal aspects to expert-curated databases, an example being the Guide to Pharmacology Database (GtoPdb) that summarises pharmacological targets and interactions [1]. Links has been used to implement a workalike version of GtoPdb [13] and we hope to build on this to provide a fully versioned implementation of GtoPdb. An important requirement here is to minimize changes to the existing system.

Finally we mention two immediate next steps. First, we plan to investigate bitemporal databases [32] allow transaction and valid time to be used together, allowing us to write quires such as “when was it recorded that Bob’s contract length was extended?” Second, at present, the result of a sequenced join must be a flat record; further work is required to understand the semantics and implementation techniques for joins that produce nested results.

8 Conclusions

In spite of decades of work on temporal databases and even an extension to the SQL standard, mainstream support for...
temporal data remains limited, requiring developers to implement temporal functionality from scratch. In this paper, we have shown how to extend language-integrated query to support transaction time and valid time data, making temporal data management accessible without explicit DBMS support. We have formalised our constructs and translational implementation strategies based on those proposed by Snodgrass [31], and proved that the translations are semantics-preserving. We have implemented our approach in the Links programming language and assessed its value through a case study. Our work is a first but significant step towards fully supporting temporal data management at the language level.

References


A Full definitions

In this appendix full definitions are provided for judgments and definitions where only selected rules could be included in the main body of the paper.

- Figure 12 presents the full definition of the typechecking judgment for $\lambda_{\text{LINQ}}$, complementing the partial definition in Figure 2.
- Figure 13 presents the full definition of the evaluation relation for $\lambda_{\text{LINQ}}$, including the rules omitted from Figure 3.
- Figure 14 shows the full typing rules for $\lambda_{\text{VLINQ}}$, complementing the selected rules in Figure 6.
- Figure 15 shows the full evaluation rules for $\lambda_{\text{VLINQ}}$, complementing the selected rules in Figure 7.
- Figure 16 presents the full translation rules for sequenced and nonsequenced operations in $\lambda_{\text{VLINQ}}$ to $\lambda_{\text{LINQ}}$, completing the rules shown in Figure 8. The rules for translating current updates are special cases of those for sequenced updates and shown in Appendix C.
### Big-step reduction rules

#### E-Val

$\text{E-Val} \quad \text{E-App} \quad \text{E-Op} \quad \text{E-If}

- $E \downarrow_{\Lambda, V} (V, \Delta)
- N \downarrow_{\Lambda, \delta} (V, \Delta)
- M \downarrow_{\Lambda, \delta} (\lambda x.L, \Delta)
- M_1 \downarrow_{\Lambda, \delta} (\lambda x.L, \Delta_1)

#### E-Iff

- $E \iff L \downarrow_{\Lambda, \delta} (\text{false}, \Delta_1)
- L \downarrow_{\Lambda, \delta} (\text{true}, \Delta_1)

#### E-Bag

- $M \downarrow_{\Lambda, \delta} (V, \Delta)
- (\gamma \downarrow_{\Lambda, \delta} (V, \Delta') \quad \text{E-App}
- N \downarrow_{\Lambda, \delta} (V, \Delta')
- M \downarrow_{\Lambda, \delta} (\lambda x.L, \Delta)

#### E-Record

- $M \uparrow_{\Lambda, \delta} (V, \Delta)
- M \uparrow_{\Lambda, \delta} (\lambda x.L, \Delta)

#### E-Project

- $M_1 \downarrow_{\Lambda, \delta} (\lambda x.L_1, \Delta_1, \Delta)
- M_2 \downarrow_{\Lambda, \delta} (\lambda x.L_2, \Delta_2)

#### E-Query

- $M \downarrow_{\Lambda, \delta} (\{\}, \Delta)
- \text{E-Query} \quad \text{E-For}
- \text{E-Insert}
- \text{E-ForEmpty}

#### E-Update

- $M \downarrow_{\Lambda, \delta} (V, \Delta)
- M' \downarrow_{\Lambda, \delta} (\lambda x.L')

#### E-Delete

- $M \downarrow_{\Lambda, \delta} (\lambda x.L, \Delta)
- M \downarrow_{\Lambda, \delta} (\lambda x.L, \Delta)

### Figure 12. Typing rules for $\lambda_{\text{LINQ}}$

### Figure 13. Semantics of $\lambda_{\text{LINQ}}$
Typing rules

\[
\begin{align*}
TV\text{-}\text{GET} & : \\ \Gamma \vdash M:\text{Table} (A) \rightarrow E & \quad \Gamma \vdash \text{get} M : \text{Bag} (\text{ValidTime} (A)) \rightarrow \{ \text{read} \} \cup E \\
TV\text{-}\text{NONSEQDELETE} & : \\ \Gamma \vdash L : \text{Table} (A) \rightarrow E & \quad A = (a_i : B_i)_{i \in I} \\
& \quad \Gamma, x : \text{ValidTime} (A) \vdash M : ! E \\
& \quad \Gamma, x : \text{ValidTime} (A) \vdash N : ! E \\
& \quad (j \in I \land \Gamma, x : \text{ValidTime} (A) \vdash N_j : \text{Time} ! \emptyset) \\
& \quad (j \in I \land \Gamma, x : \text{ValidTime} (A) \vdash N_j : \text{Time} ! \emptyset) \\
\Gamma \vdash \text{delete nonsequenced} (x \triangleq M) \text{ where } N : E \rightarrow \{ \text{write} \} \cup E \\
\end{align*}
\]

\[
\begin{align*}
TV\text{-}\text{SEQINSERT} & : \\ \Gamma \vdash M : \text{Table} (A) \rightarrow E & \quad \Gamma \vdash \text{insert sequenced} M \vdash N : E \rightarrow \{ \text{write} \} \cup E \\
TV\text{-}\text{SEQUPDATE} & : \\ \Gamma \vdash L : \text{Table} (A) \rightarrow E & \quad A = (a_i : B_i)_{i \in I} \\
& \quad \Gamma \vdash M : \text{Time} ! \emptyset \\
& \quad \Gamma \vdash M : \text{Time} ! \emptyset \\
\Gamma, x : A \vdash M : ! E \\
& \quad \Gamma \vdash L : \text{Table} (A) \rightarrow E \\
& \quad \Gamma \vdash M : \text{Time} ! \emptyset \\
& \quad \Gamma \vdash M : \text{Time} ! \emptyset \\
& \quad \Gamma, x : A \vdash M : ! E \\
\Gamma \vdash \text{update sequenced} (x \triangleq L) \text{ between } M_i \text{ and } M_j \text{ where } N : E \rightarrow \{ \text{write} \} \cup E \\
\end{align*}
\]

\[
\begin{align*}
TV\text{-}\text{NONSEQUPDATE} & : \\ \Gamma \vdash L : \text{Table} (A) \rightarrow E & \quad A = (a_i : B_i)_{i \in I} \\
& \quad \Gamma \vdash M : \text{Time} ! \emptyset \\
& \quad \Gamma \vdash M : \text{Time} ! \emptyset \\
\Gamma, x : A \vdash M : ! E \\
\Gamma, x : A \vdash M : ! E \\
\Gamma \vdash L : \text{Table} (A) \rightarrow E \\
\Gamma \vdash M : \text{Time} ! \emptyset \\
\Gamma \vdash M : \text{Time} ! \emptyset \\
\Gamma, x : A \vdash M : ! E \\
\Gamma \vdash \text{update nonsequenced} (x \triangleq L) \text{ where } M \text{ set } (a_i = N_j)_{j \in J} \text{ valid from } N' \text{ to } N'' \text{ } \{ \text{write} \} \cup E \\
\end{align*}
\]

\[
\begin{align*}
TV\text{-}\text{DELETE} & : \\ \Gamma \vdash M : \text{Table} (A) \rightarrow E & \quad \Gamma \vdash \text{delete sequenced} (x \triangleq L) \text{ between } M_i \text{ and } M_j \text{ where } N : E \rightarrow \{ \text{write} \} \cup E \\
\end{align*}
\]

\[
\begin{align*}
\end{align*}
\]

**Figure 14.** Typing rules for $\lambda_{\text{VLINQ}}$
Reduction rules

<table>
<thead>
<tr>
<th>EV-Row</th>
<th>EV-SeqDelete</th>
<th>EV-SeqInsert</th>
</tr>
</thead>
</table>
| \[
M \downarrow_{\lambda_{\text{Row}}} (V_1, \Delta_1) \]
| \[
M \downarrow_{\lambda_{\text{SeqDelete}}} (t, \Delta_1) \]
| \[
M \downarrow_{\lambda_{\text{SeqInsert}}} (t, \Delta_1) \]

**EV-Row**

\[
M \downarrow_{\lambda_{\text{Row}}} (V_1, \Delta_1) = \begin{cases} 
M \downarrow_{\lambda_{\text{Row}}} (V_2, \Delta_2) & \text{if } \text{data}_{\text{start}} \in V, \text{start} < \text{end} \\
M \downarrow_{\lambda_{\text{Row}}} (V_3, \Delta_3) & \text{otherwise}
\end{cases}
\]

**EV-SeqDelete**

\[
del(v_{\text{start}}, end) \triangleq \begin{cases} 
M \downarrow_{\lambda_{\text{SeqDelete}}} (t, \Delta_1) & \text{if } N \{v/x\} \downarrow_{\lambda_{\text{SeqDelete}}} \text{true and } \text{start} < \text{end} \\
M \downarrow_{\lambda_{\text{SeqDelete}}} (t, \Delta_1) & \text{if } N \{v/x\} \downarrow_{\lambda_{\text{SeqDelete}}} \text{true and } \text{Vstart} > \text{start} \text{ and } \text{Vend} < \text{end}
\end{cases}
\]

**EV-SeqInsert**

\[
M \downarrow_{\lambda_{\text{SeqInsert}}} (t, \Delta_1) = \begin{cases} 
M \downarrow_{\lambda_{\text{SeqInsert}}} (t, \Delta_1) & \text{if } V_{\text{start}} \leq V_{\text{end}} \\
M \downarrow_{\lambda_{\text{SeqInsert}}} (t, \Delta_1) & \text{if } V_{\text{start}} > V_{\text{end}} \text{ and } V_{\text{start}} \leq \text{end}
\end{cases}
\]

**EV-NonseqDelete**

\[
M \downarrow_{\lambda_{\text{NonseqDelete}}} (t, \Delta_1) = \begin{cases} 
M \downarrow_{\lambda_{\text{NonseqDelete}}} (t, \Delta_1) & \text{if } d \in \Delta(t) \text{ and } N \{d/x\} \downarrow_{\lambda_{\text{NonseqDelete}}} \text{false}
\end{cases}
\]

**EV-NonseqUpdate**

\[
\text{upd}(D = v_{\text{start}}, end) \triangleq \begin{cases} 
L \downarrow_{\lambda_{\text{SeqUpdate}}} (t, \Delta_1) & \text{if } N_1 \{D/x\} \downarrow_{\lambda_{\text{SeqUpdate}}} \text{true and } N_2 \{D/x\} \downarrow_{\lambda_{\text{SeqUpdate}}} \text{false}
\end{cases}
\]

**EV-SeqUpdate**

\[
\text{upd}(v_{\text{start}}, end) \triangleq \begin{cases} 
W \downarrow_{\lambda_{\text{SeqUpdate}}} (t, \Delta_1) & \text{if } M \{v/x\} \downarrow_{\lambda_{\text{SeqUpdate}}} \text{true and } V_{\text{start}} \leq \text{start} \text{ and } V_{\text{end}} \geq \text{end}
\end{cases}
\]

**Figure 15.** Reduction rules for \(\lambda_{\text{VLINQ}}\)
\<delete^{(t_1 \setminus A_1)}\> nonsequenced \((x \equiv M)\) where \(N\) =
\begin{align*}
\text{delete} & \quad (x \equiv \langle M \rangle) \quad \text{where lift}(x, \langle N \rangle) \\
\text{where lift} & \quad (x, f) = \\
\lambda x. f & \quad (data = \eta(x, \{t_i\}_{i=1}^j), \text{start} = x.\text{start}, \text{end} = x.\text{end})
\end{align*}
\begin{align*}
\langle update^{(t_1 \setminus A_1)} \rangle \text{ct} & \quad \text{nonsequenced} \quad (x \equiv L) \quad \text{where} \quad \mathcal{M} \text{ set} \ (t_j = N_j)_{j \in J} \quad \text{valid from} \quad N'_1 \text{ to } N'_2 = \\
\langle update^{(t_1 \setminus A_1)} \rangle \text{ct} & \quad (x \equiv \langle L \rangle) \\
\text{where} & \quad (\text{lift}(x, \langle M \rangle)) \\
\text{set} & \quad ((t_j = \text{lift}(x, \langle N''_j \rangle))_{j \in J}, \text{start} = \text{lift}(x, \langle N'_1 \rangle)), \text{end} = \text{lift}(x, \langle N'_2 \rangle)) \\
\text{where lift} & \quad (x, f) = \\
\lambda x. f & \quad (data = \eta(x, \{t_i\}_{i=1}^j), \text{start} = x.\text{start}, \text{end} = x.\text{end})
\end{align*}
\begin{align*}
\langle insert^{(t_1 \setminus A_1)} \rangle \text{ ct} & \quad \text{sequenced} \quad M \text{ values} \quad N \rangle = \\
\text{let} & \quad tbl = \langle M \rangle \space in \\
\text{let} & \quad \text{rows} = \\
\text{for} & \quad (x \equiv \langle N \rangle) \\
\eta(x, \bar{f}) & \quad \oplus (\text{start} = x.\text{start}, \text{end} = x.\text{end}) \\
\text{in} & \\
\text{insert tbl values} & \space \text{rows}
\end{align*}
\begin{align*}
\langle delete^{(t_1 \setminus A_1)} \rangle \text{ ct} & \quad \text{sequenced} \quad (x \equiv L) \quad \text{between} \quad M_1 \text{ and } M_2 \quad \text{where} \quad N \rangle = \\
\text{let} & \quad tbl = \langle L \rangle \space in \\
\text{let} & \quad a\text{Start} = \langle M_1 \rangle \space in \\
\text{let} & \quad a\text{End} = \langle M_2 \rangle \space in \\
\text{let} & \quad \text{StartRows} = \text{startRows}(tbl, \text{pred}, a\text{Start}, \bar{f}) \space in \\
\text{let} & \quad \text{EndRows} = \text{endRows}(tbl, \text{pred}, a\text{End}, \bar{f}) \space in \\
\text{delete} & \quad (x \equiv tbl) \\
\text{where} & \quad (\text{pred} \land (x.\text{start} < a\text{End}) \land (x.\text{end} > a\text{Start})) \\
\text{insert tbl values} & \space \text{StartRows}; \\
\text{insert tbl values} & \space \text{EndRows} \\
\text{where pred} & \quad \triangleq \text{restrict}(x, \bar{f}, \langle N \rangle)
\end{align*}
\begin{align*}
\langle update^{(t_1 \setminus A_1)} \rangle \text{ ct} & \quad \text{sequenced} \quad (x \equiv L) \quad \text{between} \quad M_1 \text{ and } M_2 \quad \text{where} \quad M_3 \text{ set} \ (t_j = N_j)_{j \in J} = \\
\text{let} & \quad tbl = \langle L \rangle \space in \\
\text{let} & \quad a\text{Start} = \langle M_1 \rangle \space in \\
\text{let} & \quad a\text{End} = \langle M_2 \rangle \space in \\
\text{let} & \quad \text{StartRows} = \text{startRows}(tbl, \text{pred}, a\text{Start}, \{t_i\}_{i=1}^j) \space in \\
\text{let} & \quad \text{EndRows} = \text{endRows}(tbl, \text{pred}, a\text{End}, \{t_i\}_{i=1}^j) \space in \\
\text{update} & \quad (x \equiv tbl) \\
\text{where} & \quad (\text{pred} \land (x.\text{start} < a\text{End}) \land (x.\text{end} > a\text{Start})) \\
\text{set} & \quad ((t_j = \text{restrict}(x, \{t_i\}_{i=1}^j, \langle N \rangle))_{j \in J}, \text{start} = \text{greatest}(x.\text{start}, a\text{Start}), \text{end} = \text{least}(x.\text{end}, a\text{End}); \\
\text{insert tbl values} & \space \text{StartRows}; \\
\text{insert tbl values} & \space \text{EndRows} \\
\text{where pred} & \quad \triangleq \text{restrict}(x, \{t_i\}_{i=1}^j, \langle M_3 \rangle)
\end{align*}

\begin{align*}
\text{startRows}(tbl, \text{pred}, a\text{Start}, \bar{f}) & \quad \triangleq \\
\text{query} & \quad (x \leftarrow \text{get tbl}) \\
\text{where} & \quad (\text{pred} \land (x.\text{start} < a\text{Start}) \land (x.\text{end} > a\text{Start})) \\
\eta(x, \bar{f}) & \quad \oplus (\text{start} = x.\text{start}, \text{end} = a\text{Start})
\end{align*}
\begin{align*}
\text{endRows}(tbl, \text{pred}, a\text{End}, \bar{f}) & \quad \triangleq \\
\text{query} & \quad (x \leftarrow \text{get tbl}) \\
\text{where} & \quad (\text{pred} \land (x.\text{start} < a\text{End}) \land (x.\text{end} > a\text{End})) \\
\eta(x, \bar{f}) & \quad \oplus (\text{start} = a\text{End}, \text{end} = x.\text{end})
\end{align*}

**Figure 16.** Translation from \(\lambda_{\text{VLINQ}}\) into \(\lambda_{\text{LINQ}}\)
### Illustration of sequenced delete and update behavior

Figure 17 illustrates graphically the five different overlap relationships that can hold between the period of validity of an existing row to be updated or deleted (PV) and the period of applicability of a deletion or update. The five cases involve when one interval is totally contained in the other (1, 3), when there is overlap but neither containment relationship holds (2, 4) and when the intervals are disjoint (5). In each case the effect of a deletion or update is described in the corresponding column of the table; generally the result is to replace the input row with zero, one, two or (exceptionally) three new rows. When the PV and PA intervals are disjoint no action needs to be taken.

<table>
<thead>
<tr>
<th>Case</th>
<th>Diagram</th>
<th>Delete behavior</th>
<th>Update behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="#" alt="Diagram 1" /></td>
<td>the entire row will be deleted</td>
<td>the entire row will be updated and adjusted to cover PA</td>
</tr>
<tr>
<td>2</td>
<td><img src="#" alt="Diagram 2" /></td>
<td>the overlapping portion will be deleted, leaving the rest of PV alone</td>
<td>insert new row with updated values covering PA and shorten existing row to only cover remainder of PV</td>
</tr>
<tr>
<td>3</td>
<td><img src="#" alt="Diagram 3" /></td>
<td>split row into two, one covering the part of PV before PA and the other covering the part of PV after PA</td>
<td>split row into three, two as in the case of deletion and a third providing the updated values for PA</td>
</tr>
<tr>
<td>4</td>
<td><img src="#" alt="Diagram 4" /></td>
<td>symmetric with case 2</td>
<td>symmetric with case 2</td>
</tr>
<tr>
<td>5</td>
<td><img src="#" alt="Diagram 5" /></td>
<td>no effect</td>
<td>no effect</td>
</tr>
</tbody>
</table>

**Figure 17.** Timeline diagrams illustrating sequenced delete and update behavior
C Direct semantics and translations for $\lambda_{VLINQ}$ current updates and deletions

In Section 4, we showed that current updates and deletions in $\lambda_{VLINQ}$ are special cases of sequenced updates and deletions. For the sake of completeness, we include the direct semantics and translations here.

**EV-Update**

$$L \downarrow_{\lambda}^{\Lambda'} (t, \Lambda') = \Lambda'[t \mapsto \left\{ \text{upd}(a) \mid a \in \Lambda'(t) \right\}]$$

$\text{upd}(\text{data}[\text{start}, \text{end}]) = \left\{ \begin{array}{ll}
\text{if } M[\text{data}[x]] \uparrow_{\star} \text{true} \text{ and } i \leq \text{start} \\
\text{if } M[\text{data}[x]] \uparrow_{\star} \text{true} \text{ and } i < \text{start} < i < \text{end} \\
\text{if } M[\text{data}[x]] \uparrow_{\star} \text{false or } i \geq \text{end}
\end{array} \right.$

update \((x \iff L) \text{ where } M\downarrow_{\lambda}^{\Lambda'} (\Lambda'', ())\)

**EV-Delete**

$$M \downarrow_{\lambda}^{\Lambda'} (t, \Lambda') = \Lambda'[t \mapsto \left\{ \text{del}(a) \mid a \in \Lambda'(t) \right\}]$$

$\text{del}(\text{data}[\text{start}, \text{end}]) = \left\{ \begin{array}{ll}
\text{if } N[\text{data}[x]] \uparrow_{\star} \text{true and } i \leq \text{start} \\
\text{if } N[\text{data}[x]] \uparrow_{\star} \text{true and } i < \text{start} < i < \text{end} \\
\text{if } N[\text{data}[x]] \uparrow_{\star} \text{false or } i \geq \text{end}
\end{array} \right.$

delete \((x \iff M) \text{ where } N\downarrow_{\lambda}^{\Lambda'} (\Lambda'', ())\)

Rules EV-DELETE and EV-UPDATE are different to their $\lambda_{TLINQ}$ counterparts, taking into account that a row may be valid further into the future than the current time.

Specifically, rule EV-DELETE has three cases: if the row being inspected is entirely in the future (i.e., its start field is greater than or equal to the current time), then it is deleted from the database. If the current time is greater than the start of the row, but less than the end of the row, then the row is closed off at the current time. The row is not modified if the current time is greater than the end time of the row, or the predicate evaluates to false. Current updates, described by EV-UPDATE, follow much the same pattern, but if the current time is between the start and end times of the row, then the row is split: the the part up until the current time retains the previous values, and the part from the current time until the end of the row is set to the new values.

$$\text{currentAt}(x, \text{time}) = \text{time} \geq x.\text{start} \land \text{time} < x.\text{end}$$

The meta-level currentAt function checks whether a row is valid at a particular timestamp.

Current deletions are implemented using two $\lambda_{VLINQ}$ operations: an update operation, which adjusts the end time of all matching, currently-valid rows to the current time; a delete operation, which entirely deletes all matching rows which begin after the current time.

To implement current updates, we firstly calculate the affected bag of rows (i.e., those rows which match the predicate and are valid at the current time), with their updated values. We then do three operations:
• An update, which adjusts the end time of all records which match the predicate, start before the current time, and end after the current time
• A second update, which updates all rows matching the predicate which both start and end after the current timestamp
• Finally, an insert, which inserts the contents of the affected bag.
D Proofs

Lemma D.1 (Pure comprehensions). If \( V = [V_1, \ldots, V_n] \) and we have a for comprehension for \( (x \leftarrow V) M \) such that \( \cdot \vdash \text{for} (x \leftarrow V) M : A ! \emptyset \) and \( M[V_i/x] \downarrow^* W'_i \) or \( M[V_i/x] \downarrow^*_\Delta W'_i \) for each \( i \in 1..n \), then for \( (x \leftarrow V) M \downarrow^* [W'_1, \ldots, W'_n] \).

Proof: Follows from the definition of for comprehension evaluation rules.

Lemma D.2 (Lifting of pure reduction). If \( \cdot \vdash M : A ! \emptyset \) and \( M \downarrow^* V \) or \( M \downarrow^*_\Delta V \), then \( M \downarrow^* \Delta (V, \Delta) \) for any \( \Delta \).

Proof: By induction on the derivation of \( M \downarrow^* V \).

Lemma D.3 (Correctness of \( \lambda_{LINQ} \) translation (pure reduction)). If \( M \downarrow^* V \), then \( [M] \downarrow^* [V] \).

Proof: By induction on the derivation of \( M \downarrow^* V \).

Lemma D.4. If \( \Gamma, x : (t_i : A_i)_{i \in I} \vdash M : B ! \emptyset \) and \( M \downarrow^* V \), and \( [\Gamma], x : ((t_i : [A_i])_{i \in I}) \oplus (t_j : [A'_j])_{j \in I} \vdash [M][A] ! \emptyset \), then \( \text{restrict}(x, (t_i)_{i \in I}, [M]) \downarrow^* \Delta_{A_j} [V] \).

Proof: Follows from the definition of restrict and Lemma D.3.

Lemma D.5 (Correctness of \( \lambda_{VLINQ} \) translation (pure reduction)). If \( M \downarrow^* V \), then \( \{M\} \downarrow^* \{V\} \).

Proof: By induction on the derivation of \( M \downarrow^* V \).

Lemma D.6. If \( \Gamma, x : (t_i : A_i)_{i \in I} \vdash M : B ! \emptyset \) and \( M \downarrow^* V \), and \( [\Gamma], x : ((t_i : [A_i])_{i \in I}) \oplus (t_j : [A'_j])_{j \in I} \vdash \{M\}[A] ! \emptyset \), then \( \text{restrict}(x, (t_i)_{i \in I}, \{M\}) \downarrow^* \Delta_{A_j} \{V\} \).

Proof: Follows from the definition of restrict and Lemma D.5.

Lemma D.7. If \( \Gamma \vdash V : A ! \emptyset \) where \( A = C \) or \( A = (t_i : C_i)_i \), then \( \{V\} = V \).

Proof: By induction on the derivation of \( \Gamma \vdash V : A ! \emptyset \).

Proposition D.1 (Type correctness of \( \lambda_{LINQ} \) translation). If \( \Gamma \vdash M : A ! E \) in \( \lambda_{LINQ} \), then \( [\Gamma] \vdash [M][A] ! E \) in \( \lambda_{LINQ} \).

Proof: By induction on the derivation of \( \Gamma \vdash M : A ! E \).

Proposition D.2 (Type correctness of \( \lambda_{VLINQ} \) translation). If \( \Gamma \vdash M : A ! E \) in \( \lambda_{VLINQ} \), then \( [\Gamma] \vdash \{M\}[A] ! E \) in \( \lambda_{LINQ} \).

Proof: By induction on the derivation of \( \Gamma \vdash M : A ! E \).

Theorem 3.1. If \( \cdot \vdash M : A ! E \) and \( M \downarrow^* (V, \Delta') \) where \( \text{wf}(\Delta) \) and \( \text{max}(\Delta) \leq i \), then \( [M] \downarrow^* \Delta_{A_j} ([V], \downarrow \Delta') \).

Proof: By induction on the derivation of \( M \downarrow^* V \). We show the cases for the database-relevant terms.

Case Accessor functions

Let us show the case for data; the others are similar.

Assumption:

\[
M \downarrow^*_A (V_1[V_2, V_3], \Delta')
\]

Translation:

\[
[M].\text{data}
\]

By the IH: \( [M] \downarrow^*_{\Delta_A} (\text{data} = [V_1], \text{start} = [V_2], \text{end} = [V_3]) \)

Evaluating in \( \lambda_{LINQ} \):

\[
(\text{data} = [V_1], \text{start} = [V_2], \text{end} = [V_3]).\text{data} \downarrow^*_{\Delta_A} ([V_1], \downarrow \Delta')
\]

as required.

Case get
Assumption:

\[ M \downarrow^T \Delta \leadsto (t, \Delta') \]

give
\[ M \downarrow^T \Delta \leadsto (\Delta'(t), \Delta') \]

Translation:

\[
\text{for } (x \leftarrow \text{get} \lbrack M \rbrack) \\
\{(\text{data} = (t, t_i), \text{start} = \text{x.start}, \text{end} = \text{x.end})\}
\]

Suppose \( \Delta(t) = \{(t_i = V_i)_{i \in [1..m]}, \ldots, (t_i = V_i)_{i \in [1..m]}\} \).

By the IH, \( [M] \downarrow^\Delta (t, \downarrow^\Delta) \)

Then,

\[
\downarrow^\Delta(t) = \\
\{(t_i = V_i), \ldots, t_m = [V_m], \text{start} = [W_1], \text{end} = [W_1]\}, \\
\ldots, \\
(\ell_i = [V_i], \ldots, \ell_m = [V_m], \text{start} = [W_1], \text{end} = [W_1])
\]

In \( \mathcal{L}_{\text{LINQ}} \), we have:

\[
\text{get } t \downarrow^\Delta \begin{cases} \\
(\ell_i = [V_i], \ldots, \ell_m = [V_m], \text{start} = [W_1], \text{end} = [W_1]), \ldots, \\
(\ell_i = [V_i], \ldots, \ell_m = [V_m], \text{start} = [W_1], \text{end} = [W_1])
\end{cases}
\]

By Lemmas D.1 and D.2

\[
\text{for } (x \leftarrow \text{get} \lbrack x \rbrack) \\
\{(\text{data} = (t, t_i), \text{start} = \text{x.start}, \text{end} = \text{x.end})\} \\
\downarrow^\Delta(t) = \begin{cases} \\
\{(\text{data} = (t, t_i), \ldots, t_m = [V_m], \text{start} = [W_1], \text{end} = [W_1]), \\
\ldots, \\
(\text{data} = (t, t_i), \ldots, t_m = [V_m], \text{start} = [W_1], \text{end} = [W_1])\}
\end{cases}
\]

as required.

Case insert

\[
M \downarrow^T \Delta \leadsto (t, \Delta') \\
N \downarrow^* V \\
\Delta'' = \Delta'[t \mapsto \Delta'(t) \cup vs]
\]

insert \( M \) values \( N \downarrow^T \Delta \leadsto (\Delta', \Delta'') \)

Translation:

\[ [\text{insert}^{(f; \Delta')}, M \text{ values } N] = \text{ let } \text{rows} = \]

\[ \text{ for } (x \leftarrow [N]) \\
\eta(x, t) \oplus (\text{start} = \text{now}, \text{end} = \text{forever}) \]

in

\[ \text{insert } [M] \text{ values rows} \]

By the typing rules and evaluation rule, \( V \) must be some bag \( \{(t_i = V_i), \ldots, (t_i = V_i)\} \)

By Lemma D.3 \( [N] \downarrow^* [V] \) with \( [V] = \{(t_i = [W_1]), \ldots, (t_i = [W_1])\} \)

By Lemmas D.1 and D.3, the \text{for} comprehension evaluates to \( \{V = \{(t_i = [W_1]), \text{start} = t, \text{end} = \text{forever})\} \).

Let us call this value \text{rows}.

By the IH, \( [M] \downarrow^\Delta (t, \downarrow^\Delta) \).

Thus, we evaluate

\[ \text{insert } t \text{ values rows } \downarrow^\Delta (t, \downarrow^\Delta[\text{start} \mapsto \downarrow^\Delta \cup \text{rows}]) \]

noting that \( \Delta'' = \Delta' \cup \text{rows} \), as required.

Case delete

Assumption:

\[
M \downarrow^T \Delta \leadsto (t, \Delta') \\
\Delta' = \Delta'[t \mapsto \{\text{del}(d) \mid d \in \Delta'(t)\}]
\]

del(data\{\text{start}, \text{end}\}) = \begin{cases} \\
\text{data\{\text{start}, \text{end}\}} & \text{if } \text{end} = \text{forever} \text{ and } N \{\text{data/}x\} \downarrow^* \text{true} \\
\text{data\{\text{start}, \text{end}\}} & \text{otherwise}
\end{cases}
\]

\[ \text{delete } (x \leftarrow M) \text{ where } N \downarrow^T \Delta \leadsto (\Delta', \Delta'') \]
Translation:

\[ \text{delete} (x \leftarrow M) \text{ where } N ] = \]
\[ \text{update} (x \leftarrow [M]) \]
\[ \text{ where } ((\lambda x . [N]) (\ell_t = x.f_t)) \land x.\text{end} = \text{forever} ) \]
\[ \text{ set } (\text{end} = \text{now}) \]

By the IH, \([M] \downarrow_{\Delta_t} (t, \downarrow\Lambda')\)

Suppose \(\Lambda'(t) = \{(\ell_t = V_i, t) \in I_{n1}^{W_{v1}}, \ldots, (\ell_t = V_{ni}, t) \in I_{n1}^{W_{vn}}\}\).

Then,

\[\downarrow\Lambda'(t) = \]
\[\{(\ell_t = [V_i], \ldots, \ell_t = [V_{ni}], \text{start} = [W_{i}], \text{end} = [W_{ni}]), \ldots, \]
\[\{(\ell_t = [V_i], \ldots, \ell_t = [V_{ni}], \text{start} = [W_{ni}], \text{end} = [W_{ni}])\}\]

Recall the definition of \text{update} in \(\lambda\)-LINQ:

\[L \downarrow_{\Delta_t} (t, \Delta_1) \quad \Delta_2 = \Delta_1[t \mapsto \downarrow \text{upd}(\nu) \mid \nu \in \Delta_1(t)] \]
\[\text{upd}(\nu) = \left\{ \begin{array}{ll} \{v \text{ with } \ell = W\} & \text{if } M\{v/x\} \uparrow_\ast^\ast \text{ true} \text{ and } (N\{v/x\} \uparrow_\ast \ W_i)_i \\
\{v \text{ with } \ell = W\} & \text{if } M\{v/x\} \uparrow_\ast^\ast \text{ false} \\
\end{array} \right. \]

\[\text{update} (x \leftarrow L) \text{ where } M \text{ set } (\ell = N) \downarrow_{\Delta_t} ((\ell), \Delta_2) \]

Let us reason on each row individually. Say we have a row \((\ell_t = V_i)_t \{W_{v1}, W_{vn}\}\) in \(\Lambda'\). Thus, in \(\downarrow\Lambda'\) that row will be \((\ell_t = [V_i], \ldots, \ell_t = [V_{ni}], \text{start} = [W_{i}], \text{end} = [W_{ni}]\); let us call this \(x\).

Now note that \((\ell_t = [V_i], \ldots, \ell_t = [V_{ni}], \text{start} = [W_{i}], \text{end} = [W_{ni}]\); which is equal to the translation of the data component of the record in the transaction time database.

Therefore by Lemma D.3, we have that \([N] \{(\ell_t = [V_i], \ldots, \ell_t = [V_{ni}], \text{start} = [W_{i}], \text{end} = [W_{ni}]\}; \text{ and } (N\{v/x\} \uparrow_\ast \ W_i)_i \text{ true}\). Since \([\text{forever}] = \text{forever}\) we have that \(\text{end} = \text{forever}\) if \(x.\text{end} = \text{forever}\). Thus the \text{update} affects the same rows.

We have that \(\text{now} \downarrow_\ast^\ast \text{ t}\) and thus \(((\ell_t = [V_i], \ldots, \ell_t = [V_{ni}], \text{start} = [W_{i}], \text{end} = [W_{ni}]\); \text{ with } \text{end} = \text{t}) = \downarrow (\ell_t = [V_i], (W_{ni})_i)\) for each affected record, as required.

Case update

Assumption:

\[L \downarrow_{\Delta_t} (t, \Delta') \quad \Delta'' = \Delta'[t \mapsto \downarrow \text{u}(\nu) \mid \nu \in \Delta_1(t)] \]
\[\text{u}(\text{data}^{\text{start,end}}) = \left\{ \begin{array}{ll} \{\text{data}^{\text{start,end}}, \text{data}^{\text{start,end}} \} & \text{if } M\{\text{data}/x\} \uparrow_\ast^\ast \text{ true,} \\
\{\text{data}^{\text{start,end}}\} & \text{ otherwise} \\
\end{array} \right. \]

\[\text{update} (x \leftarrow L) \text{ where } M \text{ set } (\ell = N) \downarrow_{\Delta_t} ((\ell), \Delta'') \]

Translation:

\[\text{update}^{(\ell_t \in I_t)} (x \leftarrow L) \text{ where } M \text{ set } (\ell = N_j)_j \]
Theorem 4.1. If \( \lambda \) and \( M \) are such that \( M \not\downarrow_{\Delta_{v}} (V, \Delta') \) then

\[ [M] \not\downarrow_{\Delta_{v}} ([V] \downarrow_{\lambda} V_{v} \downarrow_{\lambda} \Delta') \]

by Lemma A.3. Assume \( \lambda \) and \( M \) are such that \( \lambda \not\downarrow_{\Delta_{v}} M \) and \( M \not\downarrow_{\Delta_{v}} (V, \Delta') \). Then

\[ \Delta(\lambda') = [\lambda] \not\downarrow_{\Delta_{v}} ([V] \downarrow_{\lambda} V_{v} \downarrow_{\lambda} \Delta') \]

Suppose

\[ (\lambda_{i} = V_{i+1}^{m_{i+1}}) \}_{i \in 1..l} \]

Then

\[ \Delta(\lambda') = (\lambda_{i} = \lambda_{i+1}^{m_{i+1}}) \}_{i \in 1..l} \]

Suppose that the guard returns \( \text{true} \) for records 1...m, i.e., \( M\{\lambda_{i} = V_{i+1}\}_{i \in 1..l} \not\downarrow_{\lambda} \text{true} \), ..., \( M\{\lambda_{i} = V_{m_{i+1}}\}_{i \in 1..l} \not\downarrow_{\lambda} \text{true} \) with each \( V_{i} = \text{forever} \).

Suppose that the guard returns false for the remainder.

By Proposition D.1 and Lemma D.4 we have that

\[ [M] \{\lambda_{i} = V_{i+1}\}_{i \in 1..l} \not\downarrow_{\lambda} \text{true} \]

and \( V_{i+1} = \text{forever} \), ..., \( V_{m_{i+1}} = \text{forever} \).

Thus we have by Lemma D.1, Proposition D.1, and and Lemma D.4 that

\[ \text{affected} = \]

\[ (\lambda_{i} = V_{i+1}^{m_{i+1}}) \}_{i \in 1..l} \]

Note that \( \text{affected} = \Delta(\lambda') \).

By applying the \text{update} operation (with appropriate uses of Proposition D.1 and Lemma D.4), we have some \( \lambda'' \) with table \( t \) containing:

\[ \lambda''(\lambda') = \lambda_{i} \lambda_{i+1} = \lambda_{i+1} \lambda_{i} \]

and after the \text{insert} operation, the table stands

\[ \lambda''(\lambda') = \lambda_{i} \lambda_{i+1} = \lambda_{i+1} \lambda_{i} \]

which is equal to the database obtained by performing the \( \lambda_{\Delta_{v}} \) update, as required.

\[ \begin{array}{c}
\begin{array}{c}
\text{Theorem 4.1. If } \lambda \vdash M: A! E \text{ and } M \not\downarrow_{\Delta_{v}} (V, \Delta') \text{ for some } \omega(\Delta), \text{ then } [M] \not\downarrow_{\Delta_{v}} ([V], \Delta') \end{array}
\end{array}
\]
By Lemma D.5, \( \{N\} \downarrow^*_\Delta \sigma (\{V\}) \); since each \( V_i \) is a value of type \( \text{ValidTime}(A) \), we have that each \( V_i \) must be of the form \( W_{\text{data}}^{\text{data}} \).

Thus we have that by the definition of the translation on database rows and Lemma D.7, each \( \{V\} \) must be of the form:

\( \{data = W_{\text{data}}, \text{start} = W_{\text{start}}, \text{end} = W_{\text{end}}\} \)

By Lemma D.1 the for comprehension evaluates to a bag of the form \( (\eta(x, t)) \oplus (\text{start} = W_{\text{start}}, \text{end} = W_{\text{end}}) \).

Note that this matches the definition of \( \downarrow V_i \).

Therefore by E-INSERT:

\[
\frac{\{M\} \downarrow^*_{\Delta, i} \langle t, \downarrow \Delta_1 \rangle \quad \{N\} \downarrow^*_{\Delta, i} \langle \downarrow \Delta_1 \rangle}{\text{insert } M \text{ values } N \downarrow^*_{\Delta, i} \langle(), \downarrow \Delta_1 \rangle [t \mapsto \downarrow \Delta_1(t) \uplus \{V\}]}
\]

as required.

**Case** \( \text{EV-NonseqDelete} \)

**Assumption:**

\[
M \downarrow^*_{\Delta, i} \langle t, \downarrow \Delta_1 \rangle \quad \Delta_2 = \Delta_1 [t \mapsto \{d \in \Delta(t) \mid N\{d/x\} \downarrow^*_\Delta \text{false}]}
\]

**Translation:**

\[
\{\text{delete}(t; A_i)\} \downarrow^*_{\Delta, i} \text{nonsequenced } (x \leftarrow M) \quad \text{where } N \downarrow^*_{\Delta, i} \langle(), \downarrow \Delta_2 \rangle
\]

We want to show:

\[
M \downarrow^*_{\Delta, i} \langle t, \downarrow \Delta_1 \rangle \quad \Delta_2 = \downarrow \Delta_1 [t \mapsto \{d \in \downarrow \Delta_1(t) \mid \text{lift}(x, \{N\}\{d/x\}) \downarrow^*_\Delta \text{false}]}
\]

**Translation:**

\[
\{\text{delete}(t; A_i)\} \downarrow^*_{\Delta, i} \text{nonsequenced } (x \leftarrow M) \quad \text{where } N \downarrow^*_{\Delta, i} \langle(), \downarrow \Delta_2 \rangle
\]

By the IH we have that \( M \downarrow^*_{\Delta, i} \langle t, \downarrow \Delta_1 \rangle \).

By TV-NonseqDelete, we have that \( \Gamma, x : \text{ValidTime}(A) \vdash N:\text{bool}! \emptyset \).

Since \( N\{d/x\} \downarrow^*_\Delta \text{res} \), where \( d \) is some row \( (\ell_i = V_i)_{i [W_{\text{start}}, W_{\text{end}}]} \), it follows by Lemma D.5 that \( \{N\}\{d/x\} \downarrow^*_\Delta \text{res} \).

By Lemma D.7, \( \{\text{res} \} = \text{res} \).

Consider the definition of lift. We can show:

\[
\{N\} \lambda x. \{N\} \downarrow^*_\Delta \lambda x. \{N\} \quad \{\text{data} = \eta(x, \{\ell_i\}_i), \text{start} = x.\text{start}, \text{end} = x.\text{end}\} \downarrow^*_\Delta \{\text{data} = (\ell_i = V_i)_i, \text{start} = W_{\text{start}}, \text{end} = W_{\text{end}}\} = \{N\}\{d/x\} \downarrow^*_\Delta \text{res}
\]

since \( \{\text{data} = (\ell_i = V_i)_i, \text{start} = W_{\text{start}}, \text{end} = W_{\text{end}}\} = \{d\} \).

Therefore we know that the lifted translated predicate will evaluate to the same value as the source predicate in \( \lambda \text{VLINQ} \). It follows that the delete operation will affect the same rows, resulting in the same (flattened) database, as required.

**Case** \( \text{EV-NonseqUpdate} \)

Follows the same reasoning as \( \text{EV-NonseqDelete} \).

**Case** \( \text{EV-SeqUpdate} \)

As before, it suffices to reason about each case of the update in turn.

By the IH, we have that \( \{L\} \downarrow^*_{\Delta, i} \langle t, \downarrow \Delta' \rangle \).

By Lemma D.5, we have that:

- \( M_1 \downarrow^*_\Delta \{V_{\text{start}}\} \)
- \( M_2 \downarrow^*_\Delta \{V_{\text{end}}\} \)
and by Lemma D.7, \(\langle V_{\text{start}} \rangle = V_{\text{start}} \text{ and } \langle V_{\text{end}} \rangle = V_{\text{end}}\).

Therefore let \(a_{\text{Start}} = V_{\text{start}}\) and \(a_{\text{End}} = V_{\text{end}}\).

Since the two \texttt{insert} statements occur after the \texttt{update} statement, the records in \texttt{startRows} and \texttt{endRows} will not affect the \texttt{update}. It therefore suffices to prove the more general statement by considering a single row at a time.

Therefore, we now reason by cases on \texttt{upd} with a database of the form: \(\Delta' = [t \mapsto \{\langle [\text{start}, end] \rangle \}]\).

Since we know databases are well formed, we know that for each row, \(\text{start} < \text{end}\). By the premises, we also know that \(V_{\text{start}} < V_{\text{end}}\).

\textbf{Subcase} Case 1: \(M_{\mathcal{S}}\{v/x\} \models^* \text{true} \) and \(V_{\text{start}} \leq \text{start} \) and \(V_{\text{end}} \geq \text{end}\)

Here we have that \(V_{\text{start}} \leq \text{start} < \text{end} \leq V_{\text{end}}\).

Since \(V_{\text{start}} \leq \text{start}\), we have that \(x.\text{start} \leq V_{\text{start}}\). Therefore, \(\texttt{startRows} = \emptyset\).

Since \(V_{\text{end}} \geq \text{end}\), we have that \(x.\text{end} < V_{\text{end}}\). Therefore, \(\texttt{endRows} = \emptyset\).

The \texttt{update} predicate will match since by Proposition D.2 and Lemma D.6, \(\texttt{restrict}(x, \{f_i\}_{i \in I}, \{M_3\}) \models^* \text{true}\).

By Proposition D.2, and Lemmas D.6 and D.7, \(\texttt{restrict}(x, \{f_i\}_{i \in I}, \{N_i\}) \models^* W_j\) for all \(j \in J\).

In this case:

\begin{itemize}
  \item \textbf{greatest}((x.\text{start}, V_{\text{start}}) = x.\text{start}
  \item \textbf{least}(x.\text{end}, V_{\text{end}}) = x.\text{end}
\end{itemize}

Thus after the update and two null inserts we have:

\(\Delta'' = [t \mapsto \{(f_i = V_i)_{i \in I} \oplus (f_j = W_j)_{j \in J} \oplus (\text{start} = \text{start}, \text{end} = \text{end})\}]\),

which is equal to \(\{t \mapsto \{v \text{ with } f = W\}^{(\text{start}, \text{end})}\}\) as required.

\textbf{Subcase} Case 2: \(M_{\mathcal{S}}\{v/x\} \models^* \text{true} \) and \(V_{\text{start}} \leq \text{start} \) and \(V_{\text{end}} < \text{end}\)

In this case we have that \(V_{\text{start}} \leq \text{start} \leq V_{\text{end}} < \text{end}\).

Since \(x.\text{start} < V_{\text{start}}\), the \texttt{startRows} predicate does not hold and so \(\texttt{startRows} = \emptyset\).

However, \(x.\text{start} < V_{\text{end}}\) and \(x.\text{end} > V_{\text{end}}\), so \(\eta(x, \{f_j\}_{j \in J}) \oplus (\text{start} = V_{\text{end}}, \text{end} = \text{end}) \models^* W_j\).

Again the \texttt{update} predicate matches; this time we have:

\begin{itemize}
  \item \textbf{greatest}(\text{start}, V_{\text{start}}) = \text{start}
  \item \textbf{least}(\text{end}, V_{\text{end}}) = \text{end}
\end{itemize}

and after the update and \texttt{insert} our database is:

\(\{t \mapsto \{(f_i = V_i)_{i \in I} \oplus (f_j = W_j)_{j \in J} \oplus (\text{start} = \text{start}, \text{end} = \text{end})\}]\)

which is equal to

\(\{t \mapsto \{v \text{ with } (f_j = W_j)_{j \in J}\}^{(\text{start}, V_{\text{end}})} \oplus \{\text{start} = \text{start}, \text{end} = \text{end}\}\}\)

as required.

\textbf{Subcase} Case 3: \(M_{\mathcal{S}}\{v/x\} \models^* \text{true} \) and \(V_{\text{start}} > \text{start} \) and \(V_{\text{end}} < \text{end}\)

Here we have that \(\text{start} < V_{\text{start}} < V_{\text{end}} < \text{end}\).

By similar reasoning to the above case (we omit references to specific lemmas, which are as above), we have that:

\begin{itemize}
  \item The where clause of \texttt{startRows} evaluates to true as both \(x.\text{start} < V_{\text{start}}\) and \(x.\text{end} > V_{\text{end}}\). Thus \(\texttt{startRows} = \{v \oplus (\text{start} = \text{start}, \text{end} = \text{start})\}\).
  \item The where clause of \texttt{endRows} evaluates to true as both \(x.\text{start} < V_{\text{end}}\) and \(x.\text{end} > V_{\text{end}}\). Thus \(\texttt{endRows} = \{v \oplus (\text{start} = V_{\text{end}}, \text{end} = \text{end})\}\).
\end{itemize}

Similarly, the \texttt{update} will apply since \(x.\text{start} < V_{\text{end}}\) and \(x.\text{end} > V_{\text{start}}\).

We have in this case that \textbf{greatest}((\text{start}, V_{\text{start}})) = V_{\text{start}} \text{ and } \textbf{least}(\text{end}) = V_{\text{end}}\).

Thus after the update and two inserts, our final database is equal to:

\(\{t \mapsto \{v \text{ with } \text{etc}\}^{(V_{\text{start}}, V_{\text{end}})} \oplus \{\text{start} = \text{start}, \text{end} = \text{end}\}\}\)

as required.

\textbf{Subcase} Case 4: \(M_{\mathcal{S}}\{v/x\} \models^* \text{true} \) and \(V_{\text{start}} > \text{start} \) and \(V_{\text{end}} \geq \text{end}\)
Similar to Case 2, except there is a record produced in $startRows$ instead of $endRows$.

**Subcase** Case 5: other cases

This ‘catch all’ case boils down to three sub-subcases:

**Subsubcase** $M_3 \downarrow^* \text{false}$

In this case, the where clause of the two queries would evaluate to $\text{false}$, so we would have $startRows = endRows = \emptyset$. Furthermore, for the same reason, the update would not apply. Therefore, the database would be unaltered.

**Subcase** $V_{end} < start$

Here we have that $V_{start} < V_{end} \leq start < end$.

In this case, the where clause for $startRows$ would evaluate to $\text{false}$ as $x.start \not< V_{start}$, and the where clause for $endRows$ would evaluate to $\text{false}$ since $x.start \not< V_{end}$. Thus, $startRows = endRows = \emptyset$.

The update would not apply since $x.start \not< V_{end}$.

Therefore, the database will be unaltered.

**Subcase** $V_{start} > end$

Here we have that $start < end \leq V_{start} < V_{end}$.

Again, the where clause of $startRows$ would evaluate to $\text{false}$ since $x.end \not< V_{start}$, and the where clause of $endRows$ would evaluate to $\text{false}$ since $x.end \not< V_{end}$. Thus, $startRows = endRows = \emptyset$.

The update would not apply since $x.end \not< V_{start}$.

Therefore, the database will be unaltered.

**Case** EV-SeqDelete

Follows the same reasoning as EV-SeqUpdate. The main difference is that the final update is replaced with a delete, so where the translation results in an updated record in EV-SeqUpdate, the translation results in the absence of a row here. □
E  Links examples

New count values are added to the table with a sequenced insert, using the upload time as the start time.

```plaintext
fun insertNewData (new) {
  vt_insert sequenced covid_data
  values (subcat, weekdate, count)
  withValidity(
    (subcat = new.subcat, weekdate = new.weekdate, count = new.count),
    upload_time, forever)
}
```

Accepted updates are added using sequenced updates. Here accepted_updates is a list of updated values that have been approved by the user. For each element of the list, a sequenced update is made.

```plaintext
fun updateData (accepted_updates) {
  for (x <- accepted_updates)
    update sequenced (y <- covid_data)
    between (x.time_added, forever)
    where (x.subcat==y.subcat && x.weekdate==y.weekdate)
    set (count = x.new_value)
}
```

For update provenance queries of individual counts, a self join is computed over the subcategory and week fields of the valid time table to provide a nested result table where each count is associated with a list of count values and their associated start and end time information. This is a nonsequenced query because the time period information is explicitly added to the result table.

```plaintext
query nested {for (x <- vtCurrent(covid_data))
  [(subcat = x.subcat, weekdate = x.weekdate, count = x.count, mods =
    for (y <- covid_data)
      where (x.weekdate==vtData(y).weekdate &&
        x.subcat==vtData(y).subcat &&
        [(vtf=vtFrom(y),vtt=vtTo(y),
          count=vtData(y).count)])]}
```