# Query Shredding: Efficient Relational Evaluation of Queries over Nested Multisets

**Citation for published version:**
Cheney, J, Lindley, S & Wadler, P 2014, Query Shredding: Efficient Relational Evaluation of Queries over Nested Multisets. in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* ACM, pp. 1027-1038, The 2014 ACM SIGMOD International Conference, Snowbird, Utah, United States, 22/06/14. https://doi.org/10.1145/2588555.2612186

**Digital Object Identifier (DOI):**
10.1145/2588555.2612186

**Link:**
Link to publication record in Edinburgh Research Explorer

**Document Version:**
Peer reviewed version

**Published In:**
Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data

# Query shredding: Efficient relational evaluation of queries over nested multisets (extended version)


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

Sam Lindley
University of Edinburgh
Sam.Lindley@ed.ac.uk

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk





## ABSTRACT

Nested relational query languages have been explored extensively, and underlie industrial language-integrated query systems such as Microsoft's LINQ. However, relational databases do not natively support nested collections in query results. This can lead to major performance problems: if programmers write queries that yield nested results, then such systems typically either fail or generate a large number of queries. We present a new approach to query shredding, which converts a query returning nested data to a fixed number of SQL queries. Our approach, in contrast to prior work, handles multiset semantics, and generates an idiomatic SQL:1999 query directly from a normal form for nested queries. We provide a detailed description of our translation and present experiments showing that it offers comparable or better performance than a recent alternative approach on a range of examples.


## 1. INTRODUCTION

Databases are one of the most important applications of declarative programming techniques. However, relational databases only support queries against flat tables, while programming languages typically provide complex data structures that allow arbitrary combinations of types including nesting of collections (e.g. sets of sets). Motivated by this so-called *impedance mismatch*, and inspired by insights into language design based on monadic comprehensions [33], database researchers introduced nested relational query languages [26, 4, 5] as a generalisation of flat relational queries to allow nesting collection types inside records or other types. Several recent language designs, such as XQuery [25] and PigLatin [23], have further extended these ideas, and they have been particularly influential on language-integrated querying systems such as Kleisli [35], Microsoft's LINQ in C# and F# [22, 28, 6], Links [8, 20], and Ferry [11].

This paper considers the problem of translating nested queries over nested data to flat queries over a flat representation of nested data, or *query shredding* for short. Our motivation is to support a free combination of the features of nested relational query languages with those of high-level programming languages, particularly systems such as Links, Ferry, and LINQ. All three of these systems support queries over nested data structures (e.g. records containing nested sets, multisets/bags, or lists) in principle; however, only Ferry supports them in practice. Links and LINQ currently either reject such queries at run-time or execute them inefficiently in-memory by loading unnecessarily large amounts of data or issuing large numbers of queries (sometimes called *query storms* or *avalanches* [12] or the $N + 1$ *query problem*). To construct nested data structures while avoiding this performance penalty, programmers must currently write flat queries (e.g. loading in a superset of the needed source data) and convert the results to nested data structures. Manually reconstructing nested query results is tricky and hard to maintain; it may also mask optimisation opportunities.

In the Ferry system, Grust et al. [11, 12] have implemented shredding for nested list queries by adapting an XQuery-to-SQL translation called *loop-lifting* [15]. Loop-lifting produces queries that make heavy use of advanced On-Line Analytic Processing (OLAP) features of SQL:1999, such as `ROW_NUMBER` and `DENSE_RANK`, and to optimise these queries Ferry relies on a SQL:1999 query optimiser called Pathfinder [14].

Van den Bussche [31] proved expressiveness results showing that it is possible in principle to evaluate nested queries over *sets* via multiple flat queries. To strengthen the result, Van den Bussche's simulation eschews value invention mechanisms such as SQL:1999's `ROW_NUMBER`. The downside, however, is that the flat queries can produce results that are quadratically larger than needed to represent sets and may not preserve bag semantics. In particular, for bag semantics the size of a union of two nested sets can be quadratic in the size of the inputs — we give a concrete example in Appendix A.

Query shredding is related to the well-studied *query unnesting* problem [17, 10]. However, most prior work on unnesting only considers SQL queries that contain subqueries in WHERE clauses, not queries returning nested results; the main exception is Fegaras and Maier's work on query unnesting in a complex object calculus [10].

In this paper, we introduce a new approach to query shredding for nested multiset queries (a case not handled by prior work). Our work is formulated in terms of the Links language, but should be applicable to other language-integrated query systems, such as Ferry and LINQ, or to other complex-object query languages [10]. Figure 1 illustrates the behaviour of Links, Ferry and our approach.

We decompose the translation from nested to flat queries into a series of simpler translations. We leverage prior work on *normalisation* that translates a higher-order query to a *normal form* in which higher-order features have been eliminated [34, 7, 10]. Our algorithm operates on normal forms. We review normalisation in Section 2, and give a running example of our approach in Section 3.

Sections 4–7 present the remaining phases of our approach, which

This paper is an extended version of a paper to appear at SIGMOD 2014.

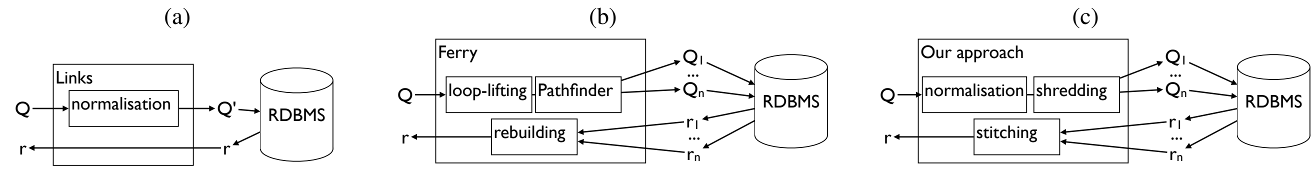


**Figure 1: (a) Default Links behaviour (flat queries) (b) Ferry (nested list queries) (c) Our approach (nested bag queries)**

are new. The *shredding* phase translates a single, nested query to a number of flat queries. These queries are organised in a *shredded package*, which is essentially a type expression whose collection type constructors are annotated with queries. The different queries are linked by *indexes*, that is, keys and foreign keys. The shredding translation is presented in Section 4. Shredding leverages the normalisation phase in that we can define translations on types and terms independently. Section 5 shows how to run shredded queries and stitch the results back together to form nested values.

The *let-insertion* phase implements a flat indexing scheme using a let-binding construct and a row-numbering operation. Let-insertion (described in Section 6) is conceptually straightforward, but provides a vital link to proper SQL by providing an implementation of abstract indexes. The final stage is to translate to SQL (Section 7) by flattening records, translating let-binding to SQL's `WITH`, and translating row-numbering to SQL's `ROW_NUMBER`. This phase also flattens nested records; the (standard) details are included in Appendix E for completeness.

We have implemented and experimentally evaluated our approach (Section 8) in comparison with Ulrich's implementation of loop-lifting in Links [30]. Our approach typically performs as well or better than loop-lifting, and can be significantly faster.

Our contribution over prior work can be summarised as follows. Fegaras and Maier [10] show how to unnest complex object queries including lists, sets, and bags, but target a nonstandard nested relational algebra, whereas we target standard SQL. Van den Bussche's simulation [31] translates nested set queries to several relational queries but was used to prove a theoretical result and was not intended as a practical implementation technique, nor has it been implemented and evaluated. Ferry [11, 12] translates nested list queries to several SQL:1999 queries and then tries to simplify the resulting queries using Pathfinder. Sometimes, however, this produces queries with cross-products inside OLAP operations such as `ROW_NUMBER`, which Pathfinder cannot simplify. In contrast, we delay introducing OLAP operations until the last stage, and our experiments show how this leads to much better performance on some queries. Finally, we handle nested multisets, not sets or lists.

This report is the extended version of a SIGMOD 2014 paper, and contains additional details, proofs of correctness, and comparison with related work in appendices.

# 2. BACKGROUND

We use metavariables $x, y, \dots, f, g$ for *variables*, and $c, d, \dots$ for *constants* and primitive operations. We also use letters $t, t', \dots$ for *table names*, $\ell, \ell', \ell_i, \dots$ for *record labels* and $a, b, \dots$ for *tags*.

We write $M[x := N]$ for capture-avoiding substitution of $N$ for $x$ in $M$. We write $\vec{x}$ for a vector $x_1, \dots, x_n$. Moreover, we extend vector notation pointwise to other constructs, writing, for example, $\langle\overrightarrow{\ell = M}\rangle$ for $\langle \ell_1 = M_1, \dots, \ell_n = M_n\rangle$.

We write: square brackets $[-]$ for the meta level list constructor; $w :: \vec{v}$ for adding the element $w$ onto the front of the list $\vec{v}$; $\vec{v} +\!\!+ \vec{w}$ for the result of appending the list $\vec{w}$ onto the end of the list $\vec{v}$; and *concat* for the function that concatenates a list of lists. We also

$$
\begin{aligned}
\mathcal{N}[\![x]\!]_\rho &= \rho(x)\\
\mathcal{N}[\![c(X_1,\dots,X_n)]\!]_\rho &= [\![c]\!](\mathcal{N}[\![X_1]\!]_\rho,\dots,\mathcal{N}[\![X_n]\!]_\rho)\\
\mathcal{N}[\![\lambda x.M]\!]_\rho &= \lambda v.\mathcal{N}[\![M]\!]_{\rho[x\mapsto v]}\\
\mathcal{N}[\![M\ N]\!]_\rho &= \mathcal{N}[\![M]\!]_\rho(\mathcal{N}[\![N]\!]_\rho)\\
\mathcal{N}[\![\langle \ell_i = M_i\rangle_{i=1}^n]\!]_\rho &= \langle \ell_i = \mathcal{N}[\![M_i]\!]_\rho\rangle_{i=1}^n\\
\mathcal{N}[\![M.\ell]\!]_\rho &= \mathcal{N}[\![M]\!]_\rho.\ell\\
\mathcal{N}[\![\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N]\!]_\rho &= \begin{cases}\mathcal{N}[\![M]\!]_\rho, & \text{if } \mathcal{N}[\![L]\!]_\rho = \mathsf{true}\\ \mathcal{N}[\![N]\!]_\rho, & \text{if } \mathcal{N}[\![L]\!]_\rho = \mathsf{false}\end{cases}\\
\mathcal{N}[\![\mathsf{return}\ M]\!]_\rho &= [\mathcal{N}[\![M]\!]_\rho]\\
\mathcal{N}[\![\emptyset]\!]_\rho &= [\,]\\
\mathcal{N}[\![M \uplus N]\!]_\rho &= \mathcal{N}[\![M]\!]_\rho +\!\!+ \mathcal{N}[\![N]\!]_\rho\\
\mathcal{N}[\![\mathsf{for}\ (x \leftarrow M)\ N]\!]_\rho &= \mathit{concat}[\mathcal{N}[\![N]\!]_{\rho[x\mapsto v]} \mid v \leftarrow \mathcal{N}[\![M]\!]_\rho]\\
\mathcal{N}[\![\mathsf{empty}\ M]\!]_\rho &= \begin{cases}\mathsf{true}, & \text{if } \mathcal{N}[\![M]\!]_\rho = [\,]\\ \mathsf{false}, & \text{if } \mathcal{N}[\![M]\!]_\rho \neq [\,]\end{cases}\\
\mathcal{N}[\![\mathsf{table}\ t]\!]_\rho &= [\![t]\!]
\end{aligned}
$$

**Figure 2: Semantics of $\lambda_{NRC}$**

make use of the following functions:

$$
\mathit{init}\ [x_i]_{i=1}^n = [x_i]_{i=1}^{n-1} \qquad \mathit{last}\ [x_i]_{i=1}^n = x_n
$$
$$
\mathit{enum}([v_1,\dots,v_m]) = [\langle 1, v_1\rangle, \dots \langle m, v_m\rangle]
$$

In the meta language we make extensive use of comprehensions, primarily list comprehensions. For instance, $[v \mid x \leftarrow xs, y \leftarrow ys, p]$, returns a copy of $v$ for each pair $\langle x, y\rangle$ of elements of $xs$ and $ys$ such that the predicate $p$ holds. We write $[v_i]_{i=1}^n$ as shorthand for $[v_i \mid 1 \le i \le n]$ and similarly, e.g., $\langle \ell_i = M_i\rangle_{i=1}^n$ for $\langle \ell_1 = M_1, \dots, \ell_n = M_n\rangle$.

## 2.1 Nested relational calculus

We take the higher-order, nested relational calculus (evaluated over bags) as our starting point. We call this $\lambda_{NRC}$; this is essentially a core language for the query components of Links, Ferry, and LINQ. The types of $\lambda_{NRC}$ include base types (integers, strings, booleans), record types $\langle\overrightarrow{\ell : A}\rangle$, bag types $\mathrm{Bag}\,A$, and function types $A \to B$.

$$
\begin{array}{ll}
\text{Types} & A, B ::= O \mid \langle\overrightarrow{\ell : A}\rangle \mid \mathrm{Bag}\,A \mid A \to B\\
\text{Base types} & O ::= \mathit{Int} \mid \mathit{Bool} \mid \mathit{String}
\end{array}
$$

We say that a type is *nested* if it contains no function types and *flat* if it contains only base and record types.

The terms of $\lambda_{NRC}$ include $\lambda$-abstractions, applications, and the standard terms of nested relational calculus.

$$
\begin{array}{lrl}
\text{Terms} & M, N ::= & x \mid c(\vec{M}) \mid \mathsf{table}\ t \mid \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ N'\\
& \mid & \lambda x.\,M \mid M\ N \mid \langle\overrightarrow{\ell = M}\rangle \mid M.\ell \mid \mathsf{empty}\ M\\
& \mid & \mathsf{return}\ M \mid \emptyset \mid M \uplus N \mid \mathsf{for}\ (x \leftarrow M)\ N
\end{array}
$$

We assume that the constants and primitive functions include boolean values with negation and conjunction, and integer values with standard arithmetic operations and equality tests. We assume special labels $\#_1, \#_2, \dots$ and encode tuple types $\langle A_1, \dots, A_n\rangle$ as record types $\langle \#_1 : A_1, \dots, \#_n : A_n\rangle$, and similarly tuple terms

$\langle M_1, \ldots, M_n \rangle$ as record terms $\langle \#_1 = M_1, \ldots, \#_n = M_n \rangle$. We assume fixed signatures $\Sigma(t)$ and $\Sigma(c)$ for tables and constants. The tables are constrained to have flat relation type ($\mathsf{Bag}\,\langle \ell_1 : O_1, \ldots, \ell_n : O_n \rangle$), and the constants must be of base type or first order $n$-ary functions ($\langle O_1, \ldots, O_n \rangle \to O$).

Most language constructs are standard. The $\emptyset$ expression builds an empty bag, $\mathsf{return}\ M$ constructs a singleton, and $M \uplus N$ builds the bag union of two collections. The $\mathsf{for}\ (x \leftarrow M)\ N$ comprehension construct iterates over a bag obtained by evaluating $M$, binds $x$ to each element, evaluates $N$ to another bag for each such binding, and takes the union of the results. The expression $\mathsf{empty}\ M$ evaluates to $\mathsf{true}$ if $M$ evaluates to an empty bag, $\mathsf{false}$ otherwise.

$\lambda_{NRC}$ employs a standard type system similar to that presented in other work [35, 20, 6]. We will also employ several typed intermediate languages and translations mapping $\lambda_{NRC}$ to SQL. All of these (straightforward) type systems are omitted due to space limits; they will be available in the full version of this paper.

*Semantics.* We give a denotational semantics in terms of lists. Though we wish to preserve bag semantics, we interpret object-level bags as meta-level lists. For meta-level values $v$ and $v'$, we consider $v$ and $v'$ equivalent as multisets if they are equal up to permutation of list elements. We use lists mainly so that we can talk about order-sensitive operations such as $\mathsf{row\_number}$.

We interpret base types as integers, booleans and strings, function types as functions, record types as records, and bag types as lists. For each table $t \in dom(\Sigma)$, we assume a fixed interpretation $[\![t]\!]$ of $t$ as a list of records of type $\Sigma(t)$. In SQL, tables do not have a list semantics by default, but we can impose one by choosing a canonical ordering for the rows of the table. We order by all of the columns arranged in lexicographic order (assuming linear orderings on field names and base types).

We assume fixed interpretations $[\![c]\!]$ for the constants and primitive operations. The semantics of nested relational calculus are shown in Figure 2. The (standard) typing rules are given in Appendix B. We let $\rho$ range over environments mapping variables to values, writing $\varepsilon$ for the empty environment and $\rho[x \mapsto v]$ for the extension of $\rho$ with $x$ bound to $v$.

## 2.2 Query normalisation

In Links, query normalisation is an important part of the execution model [7, 20]. Links currently supports only queries mapping flat tables to flat results, or *flat–flat* queries. When a subexpression denoting a query is evaluated, the subexpression is first normalised and then converted to SQL, which is sent to the database for evaluation; the tuples received in response are then converted into a Links value and normal execution resumes (see Figure1(a)).

For *flat–nested* queries that read from flat tables and produce a nested result value, our normalisation procedure is similar to the one currently used in Links [20], but we hoist all conditionals into the nearest enclosing comprehension as where clauses. This is a minor change; the modified algorithm is given in the full version of this paper. The resulting normal forms are:

$$
\begin{array}{lrcl}
\text{Query terms} & L & ::= & \biguplus \vec{C} \\
\text{Comprehensions} & C & ::= & \mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}\ M \\
\text{Generators} & G & ::= & x \leftarrow t \\
\text{Normalised terms} & M, N & ::= & X \mid R \mid L \\
\text{Record terms} & R & ::= & \langle \overrightarrow{\ell = M} \rangle \\
\text{Base terms} & X & ::= & x.\ell \mid c(\vec{X}) \mid \mathsf{empty}\ L
\end{array}
$$

Any closed flat–nested query can be converted to an equivalent term in the above normal form.

$[\![departments]\!] =$

| (id) | name |
|---|---|
| 1 | Product |
| 2 | Quality |
| 3 | Research |
| 4 | Sales |

$[\![employees]\!] =$

| (id) | dept | name | salary |
|---|---|---|---|
| 1 | Product | Alex | 20000 |
| 2 | Product | Bert | 900 |
| 3 | Research | Cora | 50000 |
| 4 | Research | Drew | 60000 |
| 5 | Sales | Erik | 2000000 |
| 6 | Sales | Fred | 700 |
| 7 | Sales | Gina | 100000 |

$[\![tasks]\!] =$

| (id) | employee | task |
|---|---|---|
| 1 | Alex | build |
| 2 | Bert | build |
| 3 | Cora | abstract |
| 4 | Cora | build |
| 5 | Cora | call |
| 6 | Cora | dissemble |
| 7 | Cora | enthuse |
| 8 | Drew | abstract |
| 9 | Drew | enthuse |
| 10 | Erik | call |
| 11 | Erik | enthuse |
| 12 | Fred | call |
| 13 | Gina | call |
| 14 | Gina | dissemble |

$[\![contacts]\!] =$

| (id) | dept | name | client |
|---|---|---|---|
| 1 | Product | Pam | false |
| 2 | Product | Pat | true |
| 3 | Research | Rob | false |
| 4 | Research | Roy | false |
| 5 | Sales | Sam | false |
| 6 | Sales | Sid | false |
| 7 | Sales | Sue | true |

**Figure 3: Sample data**

THEOREM 1. *Given a closed flat–nested query* $\vdash M : \mathsf{Bag}\ A$, *there exists a normalisation function* $norm_{\mathsf{Bag}\ A}$, *mapping each* $M$ *to an equivalent normal form* $norm_{\mathsf{Bag}\ A}(M)$.

The normalisation algorithm and correctness proof are similar to those in previous papers [7, 20, 6]. The details are given in Appendix C. The normal forms above can also be viewed as an SQL-like language allowing relation-valued attributes (similar to the complex-object calculus of Fegaras and Maier [10]). Thus, our results can also be used to support nested query results in an SQL-like query language. In this paper, however, we focus on the functional core language based on comprehensions, as illustrated in the next section.

# 3. RUNNING EXAMPLE

To motivate and illustrate our work, we present an extended example showing how our shredding translation could be used to provide useful functionality to programmers working in LINQ using F#, Ferry or Links. We first describe the code the programmer would actually write and the results the system produces. Throughout the rest of the paper, we return to this example to illustrate how the shredding translation works.

Consider a flat database schema $\Sigma$ for an organisation:

$$
\begin{array}{l}
tasks(\mathsf{employee} : String, \mathsf{task} : String) \\
employees(\mathsf{dept} : String, \mathsf{name} : String, \mathsf{salary} : Int) \\
contacts(\mathsf{dept} : String, \mathsf{name} : String, \mathsf{client} : Bool) \\
departments(\mathsf{name} : String)
\end{array}
$$

Each department has a name, a collection of employees, and a collection of external contacts. Each employee has a name, salary and a collection of tasks. Some contacts are clients. Figure 3 shows a small instance of this schema. For convenience, we also assume every table has an integer-valued key *id*.

In $\lambda_{NRC}$, queries of the form $\mathsf{for} \ldots \mathsf{where} \ldots \mathsf{return} \ldots$ are natively supported. These are *comprehensions* as found in XQuery or functional programming languages, and they generalise idiomatic

SQL `SELECT FROM WHERE` queries [4]. Unlike SQL, we can use (nonrecursive) functions to define queries with parameters, or parts of queries, and freely combine them. For example, the following functions define useful query patterns over the above schema:

$$\begin{aligned}
&\mathit{tasksOfEmp}\ e = \mathsf{for}\ (t \leftarrow \mathit{tasks})\\
&\qquad\qquad\qquad \mathsf{where}\ (t.\mathsf{employee} = e.\mathsf{name})\\
&\qquad\qquad\qquad \mathsf{return}\ t.\mathit{tsk}\\
&\mathit{contactsOfDept}\ d = \mathsf{for}\ (c \leftarrow \mathit{contacts})\\
&\qquad\qquad\qquad \mathsf{where}\ (d.\mathsf{dept} = c.\mathsf{dept})\\
&\qquad\qquad\qquad \mathsf{return}\ \langle \mathsf{name} = c.\mathsf{name}, \mathsf{client} = c.\mathsf{client}\rangle\\
&\mathit{employeesByTask}\ t =\\
&\quad \mathsf{for}\ (e \leftarrow \mathit{employees}, d \leftarrow \mathit{departments})\\
&\quad \mathsf{where}\ (e.\mathsf{name} = t.\mathsf{employee} \wedge e.\mathsf{dept} = d.\mathsf{dept})\\
&\quad \mathsf{return}\ \langle b = e.\mathsf{employee}, c = d.\mathsf{dept}\rangle
\end{aligned}$$

Nested queries allow free mixing of collection (bag) types with record or base types. For example, the following query

$$\begin{aligned}
&\mathit{employeesOfDept}\ d = \mathsf{for}\ (e \leftarrow \mathit{employees})\\
&\qquad\qquad \mathsf{where}\ (d.\mathsf{dept} = e.\mathsf{dept})\\
&\qquad\qquad \mathsf{return}\ \langle \mathsf{name} = e.\mathsf{name}, \mathsf{salary} = e.\mathsf{salary},\\
&\qquad\qquad\qquad \mathsf{tasks} = \mathit{tasksOfEmp}\ e\rangle
\end{aligned}$$

returns a nested result: a collection of employees in a department, each with an associated collection of tasks. That is, its return type is $\mathrm{Bag}\ \langle \mathsf{name}{:}\mathit{String}, \mathsf{salary}{:}\mathit{Int}, \mathsf{tasks}{:}\mathrm{Bag}\ \mathit{String}\rangle$

Consider the following nested schema for organisations:

$$\begin{aligned}
\mathit{Task} &= \mathit{String}\\
\mathit{Employee} &= \langle \mathsf{name} : \mathit{String}, \mathsf{salary} : \mathit{Int}, \mathsf{tasks} : \mathrm{Bag}\ \mathit{Task}\rangle\\
\mathit{Contact} &= \langle \mathsf{name} : \mathit{String}, \mathsf{client} : \mathit{Bool}\rangle\\
\mathit{Department} &= \langle \mathsf{name} : \mathit{String}, \mathsf{employees} : \mathrm{Bag}\ \mathit{Employee},\\
&\qquad \mathsf{contacts} : \mathrm{Bag}\ \mathit{Contact}\rangle\\
\mathit{Organisation} &= \mathrm{Bag}\ \mathit{Department}
\end{aligned}$$

Using some of the above functions we can write a query $Q_{org}$ that maps data in the flat schema $\Sigma$ to the nested type *Organisation*, as follows:

$$\begin{aligned}
Q_{org} = &\ \mathsf{for}\ (d \leftarrow \mathit{departments})\\
&\ \mathsf{return}\ (\langle \mathsf{name} = d.\mathsf{name},\\
&\qquad \mathsf{employees} = \mathit{employeesOfDept}\ d,\\
&\qquad \mathsf{contacts} = \mathit{contactsOfDept}\ d\rangle
\end{aligned}$$

We can also define and use higher-order functions to build queries, such as the following:

$$\begin{aligned}
&\mathit{filter}\ p\ xs = \mathsf{for}\ (x \leftarrow xs)\ \mathsf{where}\ (p(x))\ \mathsf{return}\ x\\
&\mathit{any}\ xs\ p = \neg(\mathsf{empty}(\mathsf{for}\ (x \leftarrow xs)\ \mathsf{where}\ (p(x))\ \mathsf{return}\ \langle\rangle))\\
&\mathit{all}\ xs\ p = \neg(\mathit{any}\ xs\ (\lambda x.\neg(p(x))))\\
&\mathit{contains}\ xs\ u = \mathit{any}\ xs\ (\lambda x.x = u)
\end{aligned}$$

To illustrate the main technical challenges of shredding, we consider a query with two levels of nesting and a union operation.

Suppose we wish to find for each department a collection of people of interest, both employees and contacts, along with a list of the tasks they perform. Specifically, we are interested in those employees that earn less than a thousand euros and those who earn more than a million euros, call them *outliers*, along with those contacts who are clients. The following code defines poor, rich, outliers, and clients:

$$\begin{aligned}
&\mathit{isPoor}\ x = x.\mathsf{salary} < 1000\\
&\mathit{isRich}\ x = x.\mathsf{salary} > 1000000\\
&\mathit{outliers}\ xs = \mathit{filter}\ (\lambda x.\mathit{isRich}\ x \vee \mathit{isPoor}\ x)\ xs\\
&\mathit{clients}\ xs = \mathit{filter}\ (\lambda x.\, x.\mathsf{client})\ xs
\end{aligned}$$

We also introduce a convenient higher-order function that uses its $f$ parameter to initialise the tasks of its elements:

$$\mathit{getTasks}\ xs\ f = \mathsf{for}\ (x \leftarrow xs)\ \mathsf{return}\ \langle \mathsf{name} = x.\mathsf{name}, \mathsf{tasks} = f\ x\rangle$$

Using the above operations, the query $Q$ returns each department, the outliers and clients associated with that department, and their tasks. We assign the special task “buy” to clients.

$$\begin{aligned}
&Q(\mathit{organisation}) =\\
&\quad \mathsf{for}\ (x \leftarrow \mathit{organisation})\\
&\quad \mathsf{return}\ (\langle \mathsf{department} = x.\mathsf{name},\\
&\qquad\quad \mathsf{people} =\\
&\qquad\qquad \mathit{getTasks}(\mathit{outliers}(x.\mathsf{employees}))\ (\lambda y.\, y.\mathsf{tasks})\\
&\qquad\quad \uplus\ \mathit{getTasks}(\mathit{clients}(x.\mathsf{contacts}))\ (\lambda y.\ \mathsf{return}\ \text{“buy”})\rangle)
\end{aligned}$$

The result type of $Q$ is:

$$\begin{aligned}
\mathit{Result} = \mathrm{Bag}\ \langle &\mathsf{department} : \mathit{String},\\
&\mathsf{people} : \mathrm{Bag}\ \langle \mathsf{name} : \mathit{String}, \mathsf{tasks} : \mathrm{Bag}\ \mathit{String}\rangle\rangle
\end{aligned}$$

We can compose $Q$ with $Q_{org}$ to form a query $Q(Q_{org})$ from the flat data stored in $\Sigma$ to the nested *Result*. The normal form of this composed query, which we call $Q_{comp}$, is as follows:

$$\begin{aligned}
Q_{comp} = &\ \mathsf{for}\ (x \leftarrow \mathit{departments})\\
&\ \mathsf{return}\ (\\
&\quad \langle \mathsf{department} = x.\mathsf{name},\\
&\quad\ \mathsf{people} =\\
&\qquad (\mathsf{for}\ (y \leftarrow \mathit{employees})\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \wedge\\
&\qquad\qquad (y.\mathsf{salary} < 1000 \vee y.\mathsf{salary} > 1000000))\\
&\qquad\ \mathsf{return}\ (\langle \mathsf{name} = y.\mathsf{name},\\
&\qquad\qquad \mathsf{tasks} = \mathsf{for}\ (z \leftarrow \mathit{tasks})\\
&\qquad\qquad\qquad \mathsf{where}\ (z.\mathsf{employee} = y.\mathsf{name})\\
&\qquad\qquad\qquad \mathsf{return}\ z.\mathsf{task}\rangle))\\
&\quad \uplus\ (\mathsf{for}\ (y \leftarrow \mathit{contacts})\\
&\qquad\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \wedge y.\mathsf{client})\\
&\qquad\ \mathsf{return}\ (\langle \mathsf{name} = y.\mathsf{name},\\
&\qquad\qquad \mathsf{tasks} = \mathsf{return}\ \text{“buy”}\rangle))\rangle)
\end{aligned}$$

The result of running $Q_{comp}$ on the data in Figure 3 is:

$$\begin{aligned}
&[\langle \mathsf{department} = \text{“Product”},\\
&\ \ \mathsf{people} = [\langle \mathsf{name} = \text{“Bert”}, \mathsf{tasks} = [\text{“build”}]\rangle,\\
&\qquad\qquad \langle \mathsf{name} = \text{“Pat”}, \mathsf{tasks} = [\text{“buy”}]\rangle]\rangle]\\
&\langle \mathsf{department} = \text{“Research”}, \mathsf{people} = \emptyset\rangle,\\
&\langle \mathsf{department} = \text{“Quality”}, \mathsf{people} = \emptyset\rangle,\\
&\langle \mathsf{department} = \text{“Sales”},\\
&\ \ \mathsf{people} = [\langle \mathsf{name} = \text{“Erik”}, \mathsf{tasks} = [\text{“call”}, \text{“enthuse”}]\rangle,\\
&\qquad\qquad \langle \mathsf{name} = \text{“Fred”}, \mathsf{tasks} = [\text{“call”}]\rangle,\\
&\qquad\qquad \langle \mathsf{name} = \text{“Sue”}, \mathsf{tasks} = [\text{“buy”}]\rangle]\rangle]
\end{aligned}$$

Now, however, we are faced with a problem: SQL databases do not directly support nested multisets (or sets). Our shredding translation, like Van den Bussche’s simulation for sets [31] and Grust et al.’s for lists [12], can translate a normalised query such as $Q_{comp} : \mathit{Result}$ that maps flat input $\Sigma$ to nested output *Result* to a fixed number of flat queries $q_1 : \mathit{Result}_1, \ldots, q_n : \mathit{Result}_n$ whose results can be combined via a *stitching* operation $Q_{stitch} : \mathit{Result}_1 \times \cdots \times \mathit{Result}_n \to \mathit{Result}$. Thus, we can simulate the query $Q_{comp}$ by running $q_1, \ldots, q_n$ remotely on the database and stitching the results together using $Q_{stitch}$. The number of intermediate queries $n$ is the *nesting degree* of *Result*, that is, the number of collection type constructors in the result type. For example, the nesting degree of $\mathrm{Bag}\ \langle A : \mathrm{Bag}\ \mathit{Int}, B : \mathrm{Bag}\ \mathit{String}\rangle$ is 3. The nesting degree of the type *Result* is also 3, which means $Q$ can be shredded into three flat queries.

The basic idea is straightforward. Whenever a nested bag appears in the output of a query, we generate an index that uniquely identifies the current context. Then a separate query produces the contents of the nested bag, where each element is paired up with its parent index. Each inner level of nesting requires a further query.

We will illustrate by showing the results of the three queries and how they can be combined to reconstitute the desired nested result.

The outer query $q_1$ contains one entry for each department, with an index $\langle a, id\rangle$ in place of each nested collection:

$$\begin{aligned} r_1 = [&\langle \text{department} = \text{“Product”},\ \text{people} = \langle a, 1\rangle\rangle,\\ &\langle \text{department} = \text{“Quality”},\ \text{people} = \langle a, 2\rangle\rangle,\\ &\langle \text{department} = \text{“Research”}, \text{people} = \langle a, 3\rangle\rangle,\\ &\langle \text{department} = \text{“Sales”},\ \ \ \ \ \text{people} = \langle a, 4\rangle\rangle] \end{aligned}$$

The second query $q_2$ generates the data needed for the people collections:

$$\begin{aligned} r_2 = [&\langle\langle a, 1\rangle, \langle \text{name} = \text{“Bert”}, \text{tasks} = \langle b, 1, 2\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \text{name} = \text{“Erik”}, \text{tasks} = \langle b, 4, 5\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \text{name} = \text{“Fred”}, \text{tasks} = \langle b, 4, 6\rangle\rangle\rangle,\\ &\langle\langle a, 1\rangle, \langle \text{name} = \text{“Pat”},\ \ \text{tasks} = \langle d, 1, 2\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \text{name} = \text{“Sue”},\ \ \text{tasks} = \langle d, 4, 7\rangle\rangle\rangle] \end{aligned}$$

The idea is to ensure that we can stitch the results of $q_1$ together with the results of $q_2$ by joining the *inner indexes* of $q_1$ (bound to the people field of each result) with the *outer indexes* of $q_2$ (bound to the first component of each result). In both cases the static components of these indexes are the same tag $a$. Joining the people field of $q_1$ to the outer index of $q_2$ correctly associates each person with the appropriate department.

Finally, let us consider the results of the innermost query $q_3$ for generating the bag bound to the tasks field:

$$\begin{aligned} r_3 = [&\langle\langle b, 1, 2\rangle, \text{“build”}\rangle, \langle\langle b, 4, 5\rangle, \text{“call”}\rangle, \langle\langle b, 4, 5\rangle, \text{“enthuse”}\rangle,\\ &\langle\langle b, 4, 6\rangle, \text{“call”}\rangle, \langle\langle d, 1, 2\rangle, \text{“buy”}\rangle, \langle\langle d, 4, 7\rangle, \text{“buy”}\rangle] \end{aligned}$$

Recall that $q_2$ returns further inner indexes for the tasks associated with each person. The two halves of the union have different static indexes for the tasks $b$ and $d$, because they arise from different comprehensions in the source term. Furthermore, the dynamic index now consists of two id fields ($x$.id and $y$.id) in each half of the union. Thus, joining the tasks field of $q_2$ to the outer index of $q_3$ correctly associates each task with the appropriate outlier.

Note that each of the queries $q_1, q_2, q_3$ produces records that contain other records as fields. This is not strictly allowed by SQL, but it is straightforward to simulate such nested records by rewriting to a query with no nested collections in its result type; this is similar to Van den Bussche's simulation [31]. However, this approach incurs extra storage and query-processing cost. Later in the paper, we explore an alternative approach which collapses the indexes at each level to a pair $\langle a, i\rangle$ of static index and a single “surrogate” integer, similarly to Ferry's approach [12]. For example, using this approach we could represent the results of $q_2$ and $q_3$ as follows:

$$\begin{aligned} r_2' = [&\langle\langle a, 1\rangle, \langle \text{name} = \text{“Bert”}, \text{tasks} = \langle b, 1\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \text{name} = \text{“Erik”}, \text{tasks} = \langle b, 2\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \text{name} = \text{“Fred”}, \text{tasks} = \langle b, 3\rangle\rangle\rangle,\\ &\langle\langle a, 1\rangle, \langle \text{name} = \text{“Pat”},\ \ \text{tasks} = \langle d, 1\rangle\rangle\rangle,\\ &\langle\langle a, 4\rangle, \langle \text{name} = \text{“Sue”},\ \ \text{tasks} = \langle d, 2\rangle\rangle\rangle]\\ r_3' = [&\langle\langle b, 1\rangle, \text{“build”}\rangle, \langle\langle b, 2\rangle, \text{“call”}\rangle, \langle\langle b, 2\rangle, \text{“enthuse”}\rangle,\\ &\langle\langle b, 3\rangle, \text{“call”}\rangle, \langle\langle d, 1\rangle, \text{“buy”}\rangle, \langle\langle d, 2\rangle, \text{“buy”}\rangle] \end{aligned}$$

The rest of this paper gives the details of the shredding translation, explains how to stitch the results of shredded queries back together, and shows how to use row_number to avoid the space overhead of indexes. We will return to the above example throughout the paper, and we will use $Q_{org}$, $Q$ and other queries based on this example in the experimental evaluation.

# 4. SHREDDING TRANSLATION

As a pre-processing step, we annotate each comprehension body in a normalised term with a unique name $a$ — the static component of an index. We write the annotations as superscripts, for example:

$$\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}^a\ M$$

In order to shred nested queries, we introduce an abstract type $\mathit{Index}$ of *indexes* for maintaining the correspondence between outer and inner queries. An index $a \diamond d$ has a static component $a$ and a dynamic component $d$. The static component $a$ links the index to the corresponding $\mathsf{return}^a$ in the query. The dynamic component identifies the current bindings of the variables in the comprehension.

Next, we modify types so that bag types have an explicit index component and we use indexes to replace nested occurrences of bags within other bags:

$$\begin{array}{lrl} \text{Shredded types} & A, B ::= & \mathrm{Bag}\,\langle \mathit{Index}, F\rangle\\ \text{Flat types} & F ::= & O \mid \langle \overrightarrow{\ell : F}\rangle \mid \mathit{Index} \end{array}$$

We also adapt the syntax of terms to incorporate indexes. After shredding, terms will have the following forms:

$$\begin{array}{lrl} \text{Query terms} & L, M ::= & \biguplus \vec{C}\\ \text{Comprehensions} & C ::= & \mathsf{return}^a\ \langle I, N\rangle\\ & \mid & \mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ C\\ \text{Generators} & G ::= & x \leftarrow t\\ \text{Inner terms} & N ::= & X \mid R \mid I\\ \text{Record terms} & R ::= & \langle \overrightarrow{\ell = N}\rangle\\ \text{Base terms} & X ::= & x.\ell \mid c(\vec{X}) \mid \mathsf{empty}\ L\\ \text{Indexes} & I, J ::= & a \diamond d\\ \text{Dynamic indexes} & d ::= & \mathsf{out} \mid \mathsf{in} \end{array}$$

A comprehension is now constructed from a sequence of generator clauses of the form $\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)$ followed by a body of the form $\mathsf{return}^a\ \langle I, N\rangle$. Each level of nesting gives rise to such a generator clause. The body always returns a pair $\langle I, N\rangle$ of an outer index $I$, denoting where the result values from the shredded query should be spliced into the final nested result, and a (flat) inner term $N$. Records are restricted to contain inner terms, which are either base types, records, or indexes, which replace nested multisets. We assume a distinguished top level static index $\top$, which allows us to treat all levels uniformly. Each shredded term is associated with an outer index out and an inner index in. (In fact out only appears in the left component of a comprehension body, and in only appears in the right component of a comprehension body. These properties will become apparent when we specify the shredding transformation on terms.)

## 4.1 Shredding types and terms

We use *paths* to point to parts of types.

$$\text{Paths}\quad p ::= \epsilon \mid {\downarrow}.p \mid \ell.p$$

The empty path is written $\epsilon$. A path $p$ can be extended by traversing a bag constructor (${\downarrow}.p$) or selecting a label ($\ell.p$). We will sometimes write $p.{\downarrow}$ for the path $p$ with $\downarrow$ appended at the end and similarly for $p.\ell$; likewise, we will write $p.\vec{\ell}$ for the path $p$ with all the labels of $\vec{\ell}$ appended. The function $\mathit{paths}(A)$ defines the set of paths to bag types in a type $A$:

$$\begin{aligned} \mathit{paths}(O) &= \{\}\\ \mathit{paths}(\langle \ell_i : A_i\rangle_{i=1}^n) &= \textstyle\bigcup_{i=1}^n \{\ell_i.p \mid p \leftarrow \mathit{paths}(A_i)\}\\ \mathit{paths}(\mathrm{Bag}\ A) &= \{\epsilon\} \cup \{{\downarrow}.p \mid p \leftarrow \mathit{paths}(A)\} \end{aligned}$$

We now define a shredding translation on types. This is defined in terms of the *inner shredding* $\llbracket A \rrbracket$, a flat type that represents the contents of a bag.

$$\begin{aligned} \llbracket O \rrbracket &= O\\ \llbracket \langle \ell_i : A_i\rangle_{i=1}^n \rrbracket &= \langle \ell_i : \llbracket A_i \rrbracket\rangle_{i=1}^n\\ \llbracket \mathrm{Bag}\ A \rrbracket &= \mathit{Index} \end{aligned}$$

$$
\begin{aligned}
\llbracket L \rrbracket_p &= \biguplus (\llbracket L \rrbracket^\star_{\top,p}) \\
\llbracket \textstyle\biguplus_{i=1}^n C_i \rrbracket^\star_{a,p} &= \mathit{concat}([\llbracket C_i \rrbracket^\star_{a,p}]_{i=1}^n) \\
\llbracket \langle \ell_i = M_i \rangle_{i=1}^n \rrbracket^\star_{a,\ell_j.p} &= \llbracket M_j \rrbracket^\star_{a,p} \\
\llbracket \mathsf{for}\,(\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}^b\ M \rrbracket^\star_{a,\epsilon} &= [\mathsf{for}\,(\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}^b\ \langle a \diamond \mathsf{out}, \lfloor M \rfloor_b \rangle] \\
\llbracket \mathsf{for}\,(\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}^b\ M \rrbracket^\star_{a,\downarrow.p} &= [\mathsf{for}\,(\vec{G}\ \mathsf{where}\ X)\ C \mid C \leftarrow \llbracket M \rrbracket^\star_{b,p}]
\end{aligned}
$$

$$
\begin{aligned}
\lfloor x.\ell \rfloor_a &= x.\ell \\
\lfloor c([X_i]_{i=1}^n) \rfloor_a &= c([\lfloor X_i \rfloor_a]_{i=1}^n) \\
\lfloor \mathsf{empty}\ L \rfloor_a &= \mathsf{empty}\ \llbracket L \rrbracket_\epsilon \\
\lfloor \langle \ell_i = M_i \rangle_{i=1}^n \rfloor_a &= \langle \ell_i = \lfloor M_i \rfloor_a \rangle_{i=1}^n \\
\lfloor L \rfloor_a &= a \diamond \mathsf{in}
\end{aligned}
$$

**Figure 4: Shredding translation on terms**

Given a path $p \in \mathit{paths}(A)$, the type $\llbracket A \rrbracket_p$ is the *outer shredding* of $A$ at $p$. It corresponds to the bag at path $p$ in $A$.

$$
\begin{aligned}
\llbracket \mathrm{Bag}\ A \rrbracket_\epsilon &= \mathrm{Bag}\,\langle \mathit{Index}, \lfloor A \rfloor \rangle \\
\llbracket \mathrm{Bag}\ A \rrbracket_{\downarrow.p} &= \llbracket A \rrbracket_p \\
\llbracket \langle \overrightarrow{\ell : A} \rangle \rrbracket_{\ell_i.p} &= \llbracket A_i \rrbracket_p
\end{aligned}
$$

For example, consider the result type $\mathit{Result}$ from Section 3. Its nesting degree is 3, and its paths are:

$$\mathit{paths}(\mathit{Result}) = \{\epsilon, \downarrow.\mathsf{people}.\epsilon, \downarrow.\mathsf{people}.\downarrow.\mathsf{tasks}.\epsilon\}$$

We can shred $\mathit{Result}$ in three ways using these three paths, yielding three shredded types:

$$
\begin{aligned}
A_1 &= \llbracket \mathit{Result} \rrbracket_\epsilon \\
A_2 &= \llbracket \mathit{Result} \rrbracket_{\downarrow.\mathsf{people}.\epsilon} \\
A_3 &= \llbracket \mathit{Result} \rrbracket_{\downarrow.\mathsf{people}.\epsilon.\downarrow.\mathsf{tasks}.\epsilon}
\end{aligned}
$$

or equivalently:

$$
\begin{aligned}
A_1 &= \mathrm{Bag}\,\langle \mathit{Index}, \langle \mathsf{department} : \mathit{String}, \mathsf{people} : \mathit{Index} \rangle \rangle \\
A_2 &= \mathrm{Bag}\,\langle \mathit{Index}, \langle \mathsf{name} : \mathit{String}, \mathsf{tasks} : \mathit{Index} \rangle \rangle \\
A_3 &= \mathrm{Bag}\,\langle \mathit{Index}, \mathit{String} \rangle
\end{aligned}
$$

The shredding translation on terms $\llbracket L \rrbracket_p$ is given in Figure 4. This takes a term $L$ and a path $p$ and gives a query $\llbracket L \rrbracket_p$ that computes a result of type $\llbracket A \rrbracket_p$, where $A$ is the type of $L$. The auxiliary translation $\llbracket M \rrbracket^\star_{a,p}$ returns the shredded comprehensions of $M$ along path $p$ with outer static index $a$. The auxiliary translation $\lfloor M \rfloor_a$ produces a flat representation of $M$ with inner static index $a$. Note that the shredding translation is linear in time and space. Observe that for emptiness tests we need only the top-level query.

Continuing the example, we can shred $Q_{comp}$ in three ways, yielding shredded queries:

$$
\begin{aligned}
q_1 &= \llbracket Q_{comp} \rrbracket_\epsilon \\
q_2 &= \llbracket Q_{comp} \rrbracket_{\downarrow.\mathsf{people}.\epsilon} \\
q_3 &= \llbracket Q_{comp} \rrbracket_{\downarrow.\mathsf{people}.\epsilon.\downarrow.\mathsf{tasks}.\epsilon}
\end{aligned}
$$

or equivalently:

$$
\begin{aligned}
q_1 = {} & \mathsf{for}\,(x \leftarrow \mathit{departments}) \\
& \quad \mathsf{return}^a\ \langle \top \diamond 1, \langle \mathsf{department} = x.\mathsf{name}, \mathsf{people} = a \diamond \mathsf{in} \rangle \rangle \\
q_2 = {} & (\mathsf{for}\,(x \leftarrow \mathit{departments}) \\
& \quad \mathsf{for}\,(y \leftarrow \mathit{employees})\ \mathsf{where}\,(x.\mathsf{name} = y.\mathsf{dept} \wedge \\
& \qquad\qquad (y.\mathsf{salary} < 1000 \vee y.\mathsf{salary} > 1000000)) \\
& \quad \mathsf{return}^b\,(\langle a \diamond \mathsf{out}, \langle \mathsf{name} = y.\mathsf{name}, \mathsf{tasks} = b \diamond \mathsf{in} \rangle \rangle)) \\
& \uplus\ (\mathsf{for}\,(x \leftarrow \mathit{departments}) \\
& \quad \mathsf{for}\,(y \leftarrow \mathit{contacts})\ \mathsf{where}\,(x.\mathsf{name} = y.\mathsf{dept} \wedge y.\mathsf{client}) \\
& \quad \mathsf{return}^d\,(\langle a \diamond \mathsf{out}, \langle \mathsf{name} = y.\mathsf{name}, \mathsf{tasks} = d \diamond \mathsf{in} \rangle \rangle)) \\
q_3 = {} & (\mathsf{for}\,(x \leftarrow \mathit{departments}) \\
& \quad \mathsf{for}\,(y \leftarrow \mathit{employees})\ \mathsf{where}\,(x.\mathsf{name} = y.\mathsf{dept} \wedge \\
& \qquad\qquad (y.\mathsf{salary} < 1000 \vee y.\mathsf{salary} > 1000000)) \\
& \quad \mathsf{for}\,(z \leftarrow \mathit{tasks})\ \mathsf{where}\,(z.\mathsf{employee} = y.\mathsf{employee}) \\
& \quad \mathsf{return}^c\ \langle b \diamond \mathsf{out}, z.\mathsf{task} \rangle) \\
& \uplus\ (\mathsf{for}\,(x \leftarrow \mathit{departments}) \\
& \quad \mathsf{for}\,(y \leftarrow \mathit{contacts})\ \mathsf{where}\,(x.\mathsf{name} = y.\mathsf{dept} \wedge y.\mathsf{client}) \\
& \quad \mathsf{return}^e\ \langle d \diamond \mathsf{out}, \text{“buy”} \rangle)
\end{aligned}
$$

As a sanity check, we show that well-formed normalised terms shred to well-formed shredded terms of the appropriate shredded types. We discuss other correctness properties of the shredding translation in Section 5. Typing rules for shredded terms are shown in Appendix B.

THEOREM 2. *Suppose $L$ is a normalised flat-nested query with $\vdash L : A$ and $p \in \mathit{paths}(A)$, then $\vdash \llbracket L \rrbracket_p : \llbracket A \rrbracket_p$.*

## 4.2 Shredded packages

To maintain the relationship between shredded terms and the structure of the nested result they are meant to construct, we use *shredded packages*. A shredded package $\hat{A}$ is a nested type with annotations, denoted $(-)^\alpha$, attached to each bag constructor.

$$\hat{A} ::= O \mid \langle \overrightarrow{\ell : \hat{A}} \rangle \mid (\mathrm{Bag}\ \hat{A})^\alpha$$

For a given package, the annotations are drawn from the same set. We write $\hat{A}(S)$ to denote a shredded package with annotations drawn from the set $S$. We sometimes omit the type parameter when it is clear from context. for shredded packages are shown in Appendix B.

Given a shredded package $\hat{A}$, we can erase its annotations to obtain its underlying type.

$$
\begin{aligned}
\mathit{erase}(O) &= O \\
\mathit{erase}(\langle \ell_i : \hat{A}_i \rangle_{i=1}^n) &= \langle \ell_i : \mathit{erase}(\hat{A}_i) \rangle_{i=1}^n \\
\mathit{erase}((\mathrm{Bag}\ \hat{A})^\alpha) &= \mathrm{Bag}\,(\mathit{erase}(\hat{A}))
\end{aligned}
$$

Given a type $A$ and a shredding function $f : \mathit{paths}(A) \to S$, we can construct a shredded package $\hat{A}(S)$.

$$
\begin{aligned}
\mathit{package}_f(A) &= \mathit{package}_{f,\epsilon}(A) \\
\mathit{package}_{f,p}(O) &= O \\
\mathit{package}_{f,p}(\langle \ell_i : A_i \rangle_{i=1}^n) &= \langle \ell_i : \mathit{package}_{f,p.\ell_i}(A_i) \rangle_{i=1}^n \\
\mathit{package}_{f,p}(\mathrm{Bag}\ A) &= (\mathrm{Bag}\,(\mathit{package}_{f,p.\downarrow}(A)))^{f(p)}
\end{aligned}
$$

Using *package*, we lift the type- and term-level shredding functions $\llbracket - \rrbracket_-$ to produce shredded packages, where each annotation contains the shredded version of the input type or query along the path to the associated bag constructor.

$$
\begin{aligned}
\mathit{shred}_B(A) &= \mathit{package}_{(\llbracket B \rrbracket_-)}(A) \\
\mathit{shred}_L(A) &= \mathit{package}_{(\llbracket L \rrbracket_-)}(A)
\end{aligned}
$$

For example, the shredded package for the $\mathit{Result}$ type from Section 3 is:

$$
\begin{aligned}
& \mathit{Shred}_{\mathit{Result}}(\mathit{Result}) = \\
& \quad \mathrm{Bag}\,\langle \mathsf{department} : \mathit{String}, \\
& \qquad\quad \mathsf{people} : \mathrm{Bag}\,\langle \mathsf{name} : \mathit{String}, \\
& \qquad\qquad\qquad \mathsf{tasks} : (\mathrm{Bag}\ \mathit{String})^{A_3} \rangle^{A_2} \rangle^{A_1}
\end{aligned}
$$

where $A_1, A_2$, and $A_3$ are as shown in Section 4.1. Shredding the normalised query $Q'$ gives the same package, except the type annotations $A_1, A_2, A_3$ become queries $q_1, q_2, q_3$.

Again, as a sanity check we show that erasure is the left inverse of type shredding and that term-level shredding operations preserve types.

THEOREM 3. *For any type $A$, we have $erase(shred_A(A)) = A$. Furthermore, if $L$ is a closed, normalised, flat–nested query such that $\vdash L : A$ then $\vdash shred_L(A) : shred_A(A)$.*

# 5. QUERY EVALUATION AND STITCHING

Having shredded a normalised nested query, we can then run all of the resulting shredded queries separately. If we stitch the shredded results together to form a nested result, then we obtain the same nested result as we would obtain by running the nested query directly. In this section we describe how to run shredded queries and stitch their results back together to form a nested result.

## 5.1 Evaluating shredded queries

The semantics of shredded queries $\mathcal{S}[\![-]\!]$ is given in Figure 5. Apart from the handling of indexes, it is much like the semantics for nested queries given in Figure 2. To allow different implementations of indexes, we first define a canonical notion of index, and then parameterise the semantics by the concrete type of indexes $X$ and a function $index : Index \to X$ mapping each canonical index to a distinct concrete index. A canonical index $a \diamond \iota$ comprises static index $a$ and dynamic index $\iota$, where the latter is a list of positive natural numbers. For now we take concrete indexes simply to be canonical indexes, and $index$ to be the identity function. We consider other definitions of $index$ in Section 6.

The current dynamic index $\iota$ is threaded through the semantics in tandem with the environment $\rho$. The former encodes the position of each of the generators in the current comprehension and allows us to invoke $index$ to construct a concrete index. The outer index at dynamic index $\iota.i$ is $\iota$; the inner index is $\iota.i$. In order for a comprehension to generate dynamic indexes we use the function $enum$ (introduced in Section 2) that takes a list of elements and returns the same list with the element number paired up with each source element.

Running a shredded query yields a list of pairs of indexes and shredded values.

$$\begin{array}{ll} \text{Results} & s ::= [\langle I_1, w_1\rangle, \ldots, \langle I_m, w_m\rangle] \\ \text{Flat values} & w ::= c \mid \langle \ell_1 = w_1, \ldots, \ell_n = w_n\rangle \mid I \end{array}$$

Given a shredded package $\hat{A}(S)$ and a function $f : S \to T$, we can map $f$ over the annotations to obtain a new shredded package $\hat{A}'(T)$ such that $erase(\hat{A}) = erase(\hat{A}')$.

$$\begin{aligned} pmap_f(O) &= O \\ pmap_f(\langle \ell_i : \hat{A}_i\rangle_{i=1}^n) &= \langle \ell_i : pmap_f(\hat{A}_i)\rangle_{i=1}^n \\ pmap_f((\mathrm{Bag}\,\hat{A})^\alpha) &= ((\mathrm{Bag}\, pmap_f(\hat{A})))^{f(\alpha)} \end{aligned}$$

The semantics of a shredded query package is a *shredded value package* containing indexed results for each shredded query. For each type $A$ we define $\mathcal{H}[\![A]\!] = shred_A(A)$ and for each flat–nested, closed $\vdash L : A$ we define $\mathcal{H}[\![L]\!]_A : \mathcal{H}[\![A]\!]$ as $pmap_{\mathcal{S}[\![-]\!]}(shred_L(A))$. In other words, we first construct the shredded query package $shred_L(A)$, then apply the shredded semantics $\mathcal{S}[\![q]\!]$ to each query $q$ in the package.

For example, here is the shredded package that we obtain after running the normalised query $Q_{comp}$ from Section 2.2:

$$\mathcal{H}[\![Q_{comp}]\!]_A = \mathrm{Bag}\,\langle \mathsf{department} : String,\ \mathsf{people} : \mathrm{Bag}\,\langle \mathsf{name} : String,\ \mathsf{tasks} : (\mathrm{Bag}\, String)^{r_3}\rangle^{r_2}\rangle^{r_1}$$

where $r_1$, $r_2$, and $r_3$ are as in Section 3 except that indexes are of the form $a \diamond 1.2.3$ instead of $\langle a, 1, 2, 3\rangle$.

## 5.2 Stitching shredded query results together

A shredded value package can be *stitched* back together into a nested value, preserving annotations, as follows:

$$\begin{aligned} stitch(\hat{A}) &= stitch_{\top \diamond 1}(\hat{A}) \\ stitch_c(O) &= c \\ stitch_r(\langle \ell_i : \hat{A}_i\rangle_{i=1}^n) &= \langle \ell_i = stitch_{r.\ell_i}(\hat{A}_i)\rangle_{i=1}^n \\ stitch_I((\mathrm{Bag}\,\hat{A})^s) &= [(stitch_w(\hat{A})) \mid \langle I, w\rangle \leftarrow s] \end{aligned}$$

The flat value parameter $w$ to the auxiliary function $stitch_w(-)$ specifies which values to stitch along the current path.

Resuming our running example, once the results $r_1 : A_1, r_2 : A_2, r_3 : A_3$ have been evaluated on the database, they are shipped back to the host system where we can run the following code in-memory to stitch the three tables together into a single value: the result of the original nested query. The code for this query $Q_{stitch}$ follows the same idea as the query $Q_{org}$ that constructs the nested $organisation$ from $\Sigma$.

$$\begin{array}{l} \mathsf{for}\ (x \leftarrow r_1) \\ \mathsf{return}\ (\langle \mathsf{department} = x.\mathsf{name}, \\ \qquad \mathsf{people} = \mathsf{for}\ (\langle i, y\rangle \leftarrow r_2) \\ \qquad\qquad \mathsf{where}\ (x.\mathsf{people} = i)) \\ \qquad\qquad \mathsf{return}\ (\langle \mathsf{name} = y.\mathsf{name}, \\ \qquad\qquad\qquad \mathsf{tasks} = \mathsf{for}\ (\langle j, z\rangle \leftarrow r_3) \\ \qquad\qquad\qquad\qquad \mathsf{where}\ (y.\mathsf{tasks} = j) \\ \qquad\qquad\qquad\qquad \mathsf{return}\ z\rangle)\rangle) \end{array}$$

We can now state our key correctness property: evaluating shredded queries and stitching the results back together yields the same results as evaluating the original nested query directly.

THEOREM 4. *If $\vdash L : \mathrm{Bag}\,A$ then:*

$$stitch(\mathcal{H}[\![L]\!]_{\mathrm{Bag}\,A}) = \mathcal{N}[\![L]\!]$$

PROOF SKETCH. We omit the full proof due to space limits; it is available in the full version of this paper. The proof introduces several intermediate definitions. Specifically, we consider an *annotated semantics* for nested queries in which each collection element is tagged with an index, and we show that this semantics is consistent with the ordinary semantics if the annotations are erased. We then prove the correctness of shredding and stitching with respect to the annotated semantics, and the desired result follows. □

# 6. INDEXING SCHEMES

So far, we have worked with canonical indexes of the form $a \diamond 1.2.3$. These could be represented in SQL by using multiple columns (padding with NULLs if necessary) since for a given query the length of the dynamic part is bounded by the number of for-comprehensions in the query. This imposes space and running time overhead due to constructing and maintaining the indexes. Instead, in this section we consider alternative, more compact indexing schemes.

We can define alternative indexing schemes by varying the $index$ parameter of the shredded semantics (see Section 5.1). Not all possible instantiations are valid. To identify those that are, we first define a function for computing the canonical indexes of a nested query result.

$$\begin{aligned} \mathcal{I}[\![L]\!] &= \mathcal{I}[\![L]\!]_{\varepsilon,1} \\ \mathcal{I}[\![\biguplus_{i=1}^n C_i]\!]_{\rho,\iota} &= concat([\mathcal{I}[\![C_i]\!]_{\rho,\iota}]_{i=1}^n) \\ \mathcal{I}[\![\langle \ell_i = M_i\rangle_{i=1}^n]\!]_{\rho,\iota} &= concat([\mathcal{I}[\![M_i]\!]_{\rho,\iota}]_{i=1}^n) \\ \mathcal{I}[\![X]\!]_{\rho,\iota} &= [\,] \end{aligned}$$

$$\begin{array}{l} \mathcal{I}[\![\mathsf{for}\ ([x_i \leftarrow t_i]_{i=1}^n\ \mathsf{where}\ X)\ \mathsf{return}^a\ M]\!]_{\rho,\iota} = \\ \quad concat([a \diamond \iota.j :: \mathcal{I}[\![M]\!]_{\rho[x_i \mapsto r_i]_{i=1}^n, \iota.j} \\ \qquad \mid \langle j, \vec{r}\rangle \leftarrow enum([\vec{r} \mid [r_i \leftarrow [\![t_i]\!]]_{i=1}^n, \mathcal{N}[\![X]\!]_{\rho[x_i \mapsto r_i]_{i=1}^n}])]) \end{array}$$

$$\mathcal{S}[\![L]\!] = \mathcal{S}[\![L]\!]_{\varepsilon,1} \qquad \mathcal{S}[\![\langle \ell = N\rangle_{i=1}^n]\!]_{\rho,\iota} = \langle \ell_i = \mathcal{S}[\![N_i]\!]_{\rho,\iota}\rangle_{i=1}^n \qquad \mathcal{S}[\![a \diamond \mathsf{out}]\!]_{\rho,\iota.i} = \mathit{index}(a \diamond \iota)$$
$$\mathcal{S}[\![X]\!]_{\rho,\iota} = \mathcal{N}[\![X]\!]_\rho \qquad \mathcal{S}[\![a \diamond \mathsf{in}]\!]_{\rho,\iota.i} = \mathit{index}(a \diamond \iota.i)$$
$$\mathcal{S}[\![\biguplus_{i=1}^n C_i]\!]_{\rho,\iota} = \mathit{concat}([\mathcal{S}[\![C_i]\!]_{\rho,\iota}]_{i=1}^n) \qquad \mathcal{S}[\![\mathsf{return}^a\, N]\!]_{\rho,\iota} = [\mathcal{S}[\![N]\!]_{\rho,\iota}]$$
$$\mathcal{S}[\![\mathsf{for}\,([x_i \leftarrow t_i]_{i=1}^n\ \mathsf{where}\ X)\, C]\!]_{\rho,\iota} = \mathit{concat}([\mathcal{S}[\![C]\!]_{\rho[x_i\mapsto r_i]_{i=1}^n,\iota.j} \mid \langle j,\vec{r}\rangle \leftarrow \mathit{enum}([\vec{r} \mid [r_i \leftarrow [\![t_i]\!]]_{i=1}^n, \mathcal{N}[\![X]\!]_{\rho[x_i\mapsto r_i]_{i=1}^n}])])$$

**Figure 5: Semantics of shredded queries**

An indexing function $\mathit{index} : \mathit{Index} \to X$ is *valid* with respect to the closed nested query $L$ if it is injective and defined on every canonical index in $\mathcal{I}[\![L]\!]$. The only requirement on indexes in the proof of Theorem 4 is that $\mathit{index}$ is valid. We consider two alternative valid indexing schemes: *natural* and *flat indexes*.

## 6.1 Natural indexes

Natural indexes are synthesised from row data. In order to generate a natural index for a query every table must have a key, that is, a collection of fields guaranteed to be unique for every row in the table. For sets, this is always possible by using all of the field values as a key; this idea is used in Van den Bussche's simulation for sets [31]. However, for bags this is not always possible, so using natural indexes may require adding extra key fields.

Given a table $t$, let $\mathit{key}_t$ be the function that given a row $r$ of $t$ returns the key fields of $r$. We now define a function to compute the list of natural indexes for a query $L$.

$$\begin{array}{c}
\mathcal{I}^\natural[\![L]\!] = \mathcal{I}^\natural[\![L]\!]_\varepsilon \\
\mathcal{I}^\natural[\![\biguplus_{i=1}^n C_i]\!]_\rho = \mathit{concat}([\mathcal{I}^\natural[\![C_i]\!]_\rho]_{i=1}^n) \\
\mathcal{I}^\natural[\![\langle \ell_i = M_i\rangle_{i=1}^n]\!]_\rho = \mathit{concat}([\mathcal{I}^\natural[\![M_i]\!]_\rho]_{i=1}^n) \\
\mathcal{I}^\natural[\![X]\!]_\rho = [\,] \\
\mathcal{I}^\natural[\![\mathsf{for}\,([x_i \leftarrow t_i]_{i=1}^n\ \mathsf{where}\ X)\ \mathsf{return}^a\ M]\!]_\rho = \\
\mathit{concat}([a \diamond \langle \mathit{key}_{t_i}(r_i)\rangle_{i=1}^n :: \mathcal{I}^\natural[\![M]\!]_{\rho[x_i\mapsto r_i]_{i=1}^n} \\
\mid [r_i \leftarrow [\![t_i]\!]]_{i=1}^n, \mathcal{N}[\![X]\!]_{\rho[x_i\mapsto r_i]_{i=1}^n}])
\end{array}$$

If $a \diamond \iota$ is the $i$-th element of $\mathcal{I}[\![L]\!]$, then $\mathit{index}_L^\natural(a \diamond \iota)$ is defined as the $i$-th element of $\mathcal{I}^\natural[\![L]\!]$. The natural indexing scheme is defined by setting $\mathit{index} = \mathit{index}_L^\natural$.

An advantage of natural indexes is that they can be implemented in plain SQL, so for a given comprehension all where clauses can be amalgamated (using the $\wedge$ operator) and no auxiliary subqueries are needed. The downside is that the type of a dynamic index may still vary across the component comprehensions of a shredded query, complicating implementation of the query (due to the need to pad some subqueries with null columns) and potentially decreasing performance due to increased data movement. We now consider an alternative, in which row_number is used to generate dynamic indexes.

## 6.2 Flat indexes and let-insertion

The idea of flat indexes is to enumerate all of the canonical dynamic indexes associated with each static index and use the enumeration as the dynamic index.

Let $\iota$ be the $i$-th element of the list $[\iota' \mid a \diamond \iota' \leftarrow \mathcal{I}[\![L]\!]]$, then $\mathit{index}_L^\flat(a \diamond \iota) = \langle a, i\rangle$. The flat indexing scheme is defined by setting $\mathit{index} = \mathit{index}_L^\flat$. Let $\mathcal{I}^\flat[\![L]\!] = [\mathit{index}_L^\flat(I) \mid I \leftarrow \mathcal{I}[\![L]\!]]$ and let $\mathcal{S}^\flat[\![L]\!]$ be $\mathcal{S}[\![L]\!]$ where $\mathit{index} = \mathit{index}_L^\flat$.

In this section, we give a translation called let-insertion that uses let-binding and an index primitive to manage flat indexes. In the next section, we take the final step from this language to SQL.

Our semantics for shredded queries uses canonical indexes. We now specify a target language providing flat indexes. In order to do so, we introduce let-bound sub-queries, and translate each comprehension into the following form:

$$\begin{array}{l}
\mathsf{let}\ q = \mathsf{for}\,(\overrightarrow{G_{\mathsf{out}}}\ \mathsf{where}\ X_{\mathsf{out}})\ \mathsf{return}\ N_{\mathsf{out}}\ \mathsf{in} \\
\quad \mathsf{for}\,(\overrightarrow{G_{\mathsf{in}}}\ \mathsf{where}\ X_{\mathsf{in}})\ \mathsf{return}\ N_{\mathsf{in}}
\end{array}$$

The special index expression is available in each loop body, and is bound to the current index value.

Following let-insertion, the types are as before, except indexes are represented as pairs of integers $\langle a, d\rangle$ where $a$ is the static component and $d$ is the dynamic component.

$$\begin{array}{ll}
\text{Types} & A, B ::= \mathrm{Bag}\,\langle\langle \mathit{Int}, \mathit{Int}\rangle, F\rangle \\
\text{Flat types} & F ::= O \mid \langle\overrightarrow{\ell : F}\rangle \mid \langle \mathit{Int}, \mathit{Int}\rangle
\end{array}$$

The syntax of terms is adapted as follows:

$$\begin{array}{ll}
\text{Query terms} & L, M ::= \biguplus \vec{C} \\
\text{Comprehensions} & C ::= \mathsf{let}\ q = S\ \mathsf{in}\ S' \\
\text{Subqueries} & S ::= \mathsf{for}\,(\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}\ N \\
\text{Data sources} & u ::= t \mid q \\
\text{Generators} & G ::= x \leftarrow u \\
\text{Inner terms} & N ::= X \mid R \mid \mathsf{index} \\
\text{Record terms} & R ::= \langle\overrightarrow{\ell = N}\rangle \\
\text{Base terms} & X ::= x.\vec{\ell} \mid c(\vec{X}) \mid \mathsf{empty}\ L
\end{array}$$

The semantics of let-inserted queries is given in Figure 6. Rather than maintaining a canonical index, it generates a flat index for each subquery.

We first give the translation on shredded types as follows:

$$\begin{array}{c}
\mathbf{L}(O) = O \\
\mathbf{L}(\langle\overrightarrow{\ell : F}\rangle) = \langle\overrightarrow{\ell : \mathbf{L}(F)}\rangle \\
\mathbf{L}(\mathit{Index}) = \langle \mathit{Int}, \mathit{Int}\rangle \\
\mathbf{L}(\mathrm{Bag}\,\langle \mathit{Index}, F\rangle) = \mathrm{Bag}\,\langle\langle \mathit{Int}, \mathit{Int}\rangle, \mathbf{L}(F)\rangle
\end{array}$$

For example:

$$\mathbf{L}(A_2) = \mathrm{Bag}\,\langle\langle \mathit{Int}, \mathit{Int}\rangle, \langle \mathsf{name} : \mathit{String}, \mathsf{tasks} : \langle \mathit{Int}, \mathit{Int}\rangle\rangle\rangle$$

Without loss of generality we rename all the bound variables in our source query to ensure that all bound variables have distinct names, and that none coincides with the distinguished name $z$ used for let-bindings. The let-insertion translation $\mathbf{L}$ is defined in Figure 7, where we use the following auxiliary functions:

$$\begin{array}{c}
\mathit{expand}(x, t) = \langle \ell_i = x.\ell_i\rangle_{i=1}^n \\
\qquad\qquad \text{where } \Sigma(t) = \mathrm{Bag}\,\langle\overrightarrow{\ell : A}\rangle \\[1ex]
\mathit{gens}\ (\mathsf{for}\,(\vec{G}\ \mathsf{where}\ X)\ C) = \vec{G} :: \mathit{gens}\ C \\
\mathit{gens}\ (\mathsf{return}^a\ N) = [\,] \\[1ex]
\mathit{conds}\ (\mathsf{for}\,(\vec{G}\ \mathsf{where}\ X)\ C) = X :: \mathit{conds}\ C \\
\mathit{conds}\ (\mathsf{return}^a\ N) = [\,] \\[1ex]
\mathit{body}\ (\mathsf{for}\,(\vec{G}\ \mathsf{where}\ X)\ C) = \mathit{body}\ C \\
\mathit{body}\ (\mathsf{return}^a\ N) = N
\end{array}$$

Each comprehension is rearranged into two sub-queries. The first generates the outer indexes. The second computes the results. The translation sometimes produces $n$-ary projections in order to refer to values bound by the first subquery inside the second.

$$
\begin{array}{llll}
\mathcal{L}[\![L]\!] = \mathcal{L}[\![L]\!]_{\varepsilon} & \mathcal{L}[\![t]\!]_{\rho} = [\![t]\!] & \mathcal{L}[\![\langle \ell_j = N_j \rangle_{j=1}^{m}]\!]_{\rho,i} = \langle \ell_j = \mathcal{L}[\![N_j]\!]_{\rho,i} \rangle_{j=1}^{m} & \\
\mathcal{L}[\![\biguplus_{j=1}^{m} C_j]\!]_{\rho} = \mathit{concat}([\mathcal{L}[\![C_j]\!]_{\rho}]_{j=1}^{m}) & \mathcal{L}[\![q]\!]_{\rho} = \rho(q) & \mathcal{L}[\![X]\!]_{\rho,i} = \mathcal{N}[\![X]\!]_{\rho} & \mathcal{L}[\![\mathsf{index}]\!]_{\rho,i} = i
\end{array}
$$

$$
\mathcal{L}[\![\mathsf{let}\ q = S_{\mathsf{out}}\ \mathsf{in}\ S_{\mathsf{in}}]\!]_{\rho} = \mathcal{L}[\![S_{\mathsf{in}}]\!]_{\rho[q \mapsto \mathcal{L}[\![S_{\mathsf{out}}]\!]_{\rho}]}
$$

$$
\mathcal{L}[\![\mathsf{for}\ ([x_j \leftarrow u_j]_{j=1}^{m}\ \mathsf{where}\ X)\ \mathsf{return}\ N]\!]_{\rho} = [\mathcal{L}[\![N]\!]_{\rho[x_j \mapsto r_j]_{j=1}^{m},i} \mid \langle i, \vec{r} \rangle \leftarrow \mathit{enum}([\vec{r} \mid [r_j \leftarrow \mathcal{L}[\![u_j]\!]_{\rho}]_{j=1}^{m}, \mathcal{N}[\![X]\!]_{\rho[x_j \mapsto r_j]_{j=1}^{m}}])]
$$

**Figure 6: Semantics of let-inserted shredded queries**

$$
\mathbf{L}(\biguplus_{i=1}^{n} C_i) = \biguplus_{i=1}^{n} \mathbf{L}(C_i)
$$

$$
\mathbf{L}(C) = \mathsf{let}\ q = (\mathsf{for}\ (\overrightarrow{G_{\mathsf{out}}}\ \mathsf{where}\ X_{\mathsf{out}})\ \mathsf{return}\ \langle R_{\mathsf{out}}, \mathsf{index} \rangle)\ \mathsf{in}\ \mathsf{for}\ (z \leftarrow q, \overrightarrow{G_{\mathsf{in}}}\ \mathsf{where}\ \mathbf{L}_{\vec{y}}(X_{\mathsf{in}}))\ \mathsf{return}\ \mathbf{L}_{\vec{y}}(N)
$$

$$
\begin{array}{lllll}
\text{where} & \overrightarrow{G_{\mathsf{out}}} = \mathit{concat}\ (\mathit{init}\ (\mathit{gens}\ C)) & \overrightarrow{y = t} = \overrightarrow{G_{\mathsf{out}}} & \overrightarrow{G_{\mathsf{in}}} = \mathit{last}\ (\mathit{gens}\ C) & N = \mathit{body}\ C \\
& X_{\mathsf{out}} = \bigwedge \mathit{init}\ (\mathit{conds}\ C) & R_{\mathsf{out}} = \langle \mathit{expand}(y_i, t_i) \rangle_{i=1}^{n} & X_{\mathsf{in}} = \mathit{last}\ (\mathit{conds}\ C) & n = \mathit{length}\ \overrightarrow{G_{\mathsf{out}}}
\end{array}
$$

$$
\mathbf{L}_{\vec{y}}(x.\ell) = \begin{cases} x.\ell, & \text{if } x \notin \{y_1, \ldots, y_n\} \\ z.1.i.\ell, & \text{if } x = y_i \end{cases}
$$

$$
\mathbf{L}_{\vec{y}}(c(X_1, \ldots, X_m)) = c(\mathbf{L}_{\vec{y}}(X_1), \ldots, \mathbf{L}_{\vec{y}}(X_m))
$$

$$
\mathbf{L}_{\vec{y}}(\langle \ell_j = X_j \rangle_{j=1}^{m}) = \langle \ell_j = \mathbf{L}_{\vec{y}}(X_j) \rangle_{j=1}^{m}
$$

$$
\mathbf{L}_{\vec{y}}(a \diamond d) = \langle a, \mathbf{L}(d) \rangle
$$

$$
\mathbf{L}_{\vec{y}}(\mathsf{empty}\ L) = \mathsf{empty}\ (\mathbf{L}_{\vec{y}}(L))
$$

$$
\mathbf{L}_{\vec{y}}(\biguplus_{i=1}^{n} C_i) = \biguplus_{i=1}^{n} \mathbf{L}_{\vec{y}}(C_i)
$$

$$
\mathbf{L}_{\vec{y}}(\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}^{a}\ \langle a, N \rangle) = \mathsf{for}\ (\vec{G}\ \mathsf{where}\ \mathbf{L}_{\vec{y}}(X))\ \mathsf{return}\ \langle a, \mathbf{L}_{\vec{y}}(N) \rangle
$$

$$
\mathbf{L}(\mathsf{out}) = z.2 \qquad \mathbf{L}(\mathsf{in}) = \mathsf{index}
$$

**Figure 7: The let-insertion translation**

For example, applying $\mathbf{L}$ to $q_1$ from Section 4.2 yields:

$$
\begin{array}{l}
\mathsf{for}\ (x \leftarrow \mathit{departments}) \\
\mathsf{return}\ \langle \langle \top, 1 \rangle, \langle \mathsf{dept} = x.\mathsf{name}, \mathsf{people} = \mathsf{index} \rangle \rangle
\end{array}
$$

and $q_2$ becomes:

$$
\begin{array}{l}
(\mathsf{let}\ q = \mathsf{for}\ (x \leftarrow \mathit{departments})\ \mathsf{return}\ \langle \langle \mathsf{dept} = x.\mathsf{name} \rangle, \mathsf{index} \rangle\ \mathsf{in} \\
\quad \mathsf{for}\ (z \leftarrow q, y \leftarrow \mathit{employees})\ \mathsf{where}\ (z.1.1.\mathsf{name} = y.\mathsf{dept} \land \\
\qquad\qquad (y.\mathsf{salary} < 1000 \lor y.\mathsf{salary} > 1000000)) \\
\quad \mathsf{return}\ (\langle \langle a, z.2 \rangle, \langle \mathsf{name} = y.\mathsf{name}, \mathsf{tasks} = \langle b, \mathsf{index} \rangle \rangle \rangle)) \\
\uplus \\
(\mathsf{let}\ q = \mathsf{for}\ (x \leftarrow \mathit{departments})\ \mathsf{return}\ \langle \langle \mathsf{dept} = x.\mathsf{name} \rangle, \mathsf{index} \rangle\ \mathsf{in} \\
\quad \mathsf{for}\ (z \leftarrow q, y \leftarrow \mathit{contacts})\ \mathsf{where}\ (z.1.1.\mathsf{name} = y.\mathsf{dept} \land y.\mathsf{client}) \\
\quad \mathsf{return}\ (\langle \langle a, z.2 \rangle, \langle \mathsf{name} = y.\mathsf{name}, \mathsf{tasks} = \langle d, \mathsf{index} \rangle \rangle \rangle))
\end{array}
$$

As a sanity check, we show that the translation is type-preserving:

THEOREM 5. *Given shredded query* $\vdash M : \mathrm{Bag}\ \langle \mathit{Index}, F \rangle$, *then* $\vdash \mathbf{L}(M) : \mathbf{L}(\mathrm{Bag}\ \langle \mathit{Index}, F \rangle)$.

To prove the correctness of let-insertion, we need to show that the shredded semantics and let-inserted semantics agree. In the statement of the correctness theorem, recall that $\mathcal{S}^{\flat}[\![L]\!]$ refers to the version of $\mathcal{S}[\![L]\!]$ using $\mathit{index} = \mathit{index}_{L}^{\flat}$.

THEOREM 6. *Suppose* $\vdash L : A$ *and* $[\![L]\!]_{p} = M$. *Then* $\mathcal{S}^{\flat}[\![M]\!] = \mathcal{L}[\![\mathbf{L}(M)]\!]$.

PROOF SKETCH. The high-level idea is to separate results into data and indexes and compare each separately. It is straightforward, albeit tedious, to show that the different definitions are equal if we replace all dynamic indexes by a dummy value. It then remains to show that the dynamic indexes agree. The pertinent case is the translation of a comprehension:

$$
[\mathsf{for}\ (\vec{G_i} \leftarrow X_i)]_{i=1}^{n}\ \mathsf{for}\ (\vec{G_{\mathsf{in}}} \leftarrow X_{\mathsf{in}})\ \mathsf{return}^{b}\ \langle a \diamond \mathsf{out}, N \rangle
$$

which becomes $\mathsf{let}\ q = S_{\mathsf{out}}\ \mathsf{in}\ S_{\mathsf{in}}$ for suitable $S_{\mathsf{out}}$ and $S_{\mathsf{in}}$. The dynamic indexes computed by $S_{\mathsf{out}}$ coincide exactly with those of $\mathcal{I}^{\flat}[\![L]\!]$ at static index $a$, and the dynamic indexes computed by $S_{\mathsf{in}}$, if there are any, coincide exactly with those of $\mathcal{I}^{\flat}[\![L]\!]$. □

## 7. CONVERSION TO SQL

Earlier translation stages have employed nested records for convenience, but SQL does not support nested records. At this stage, we eliminate nested records from queries. For example, we can represent a nested record $\langle \mathsf{a} = \langle \mathsf{b} = 1, \mathsf{c} = 2 \rangle, \mathsf{d} = 3 \rangle$ as a flat record $\langle \mathsf{a\_b} = 1, \mathsf{a\_c} = 2, \mathsf{d} = 3 \rangle$. The (standard) details are presented in Appendix E.

In order to interpret shredded, flattened, let-inserted terms as SQL, we interpret index generators using SQL's OLAP facilities.

| | |
|---|---|
| Query terms | $L ::= (\mathsf{union\ all})\ \vec{C}$ |
| Comprehensions | $C ::= \mathsf{with}\ q\ \mathsf{as}\ (S)\ C \mid S'$ |
| Subqueries | $S ::= \mathsf{select}\ R\ \mathsf{from}\ \vec{G}\ \mathsf{where}\ X$ |
| Data sources | $u ::= t \mid q$ |
| Generators | $G ::= u\ \mathsf{as}\ x$ |
| Inner terms | $N ::= X \mid \mathsf{row\_number}()\ \mathsf{over}\ (\mathsf{order\ by}\ \vec{X})$ |
| Record terms | $R ::= \overrightarrow{N\ \mathsf{as}\ \ell}$ |
| Base terms | $X ::= x.\ell \mid c(\vec{X}) \mid \mathsf{empty}\ L$ |

Continuing our example, $\mathbf{L}(q_1)$ and $\mathbf{L}(q_2)$ translate to $q_1'$ and $q_2'$ where:

$$
\begin{array}{ll}
q_1' = & \mathsf{select}\ x.\mathsf{name}\ \mathsf{as}\ \mathsf{i1\_name}, \\
& \quad \mathsf{row\_number}()\ \mathsf{over}\ (\mathsf{order\ by}\ x.\mathsf{name})\ \mathsf{as}\ \mathsf{i1\_people} \\
& \mathsf{from}\ \mathit{departments}\ \mathsf{as}\ x \\
q_2' = & (\mathsf{with}\ q\ \mathsf{as}\ (\mathsf{select}\ x.\mathsf{name}\ \mathsf{as}\ \mathsf{i1\_name}, \\
& \qquad \mathsf{row\_number}()\ \mathsf{over}\ (\mathsf{order\ by}\ x.\mathsf{name})\ \mathsf{as}\ \mathsf{i2} \\
& \qquad \mathsf{from}\ \mathit{departments}\ \mathsf{as}\ x) \\
& \ \mathsf{select}\ a\ \mathsf{as}\ \mathsf{i1\_1}, z.\mathsf{i2}\ \mathsf{as}\ \mathsf{i1\_2}, y.\mathsf{name}\ \mathsf{as}\ \mathsf{i2\_name}, b\ \mathsf{as}\ \mathsf{i2\_tasks\_1}, \\
& \qquad \mathsf{row\_number}()\ \mathsf{over}\ (\mathsf{order\ by}\ z.\mathsf{i1\_name}, z.\mathsf{i2}, \\
& \qquad\qquad y.\mathsf{dept}, y.\mathsf{employee}, y.\mathsf{salary})\ \mathsf{as}\ \mathsf{i2\_tasks\_2} \\
& \ \mathsf{from}\ \mathit{employees}\ \mathsf{as}\ y,\ q\ \mathsf{as}\ z \\
& \ \mathsf{where}\ (z.\mathsf{i1\_name} = y.\mathsf{dept} \land \\
& \qquad\qquad (y.\mathsf{salary} < 1000 \lor y.\mathsf{salary} > 1000000))) \\
& \mathsf{union\ all} \\
& (\mathsf{with}\ q\ \mathsf{as}\ (\mathsf{select}\ x.\mathsf{name}\ \mathsf{as}\ \mathsf{i1\_name}, \\
& \qquad \mathsf{row\_number}()\ \mathsf{over}\ (\mathsf{order\ by}\ x.\mathsf{name})\ \mathsf{as}\ \mathsf{i2} \\
& \qquad \mathsf{from}\ \mathit{departments}\ \mathsf{as}\ x) \\
& \ \mathsf{select}\ a\ \mathsf{as}\ \mathsf{i1\_1}, z.\mathsf{i2}\ \mathsf{as}\ \mathsf{i1\_2}, y.\mathsf{name}\ \mathsf{as}\ \mathsf{i2\_name}, d\ \mathsf{as}\ \mathsf{i2\_tasks\_1}, \\
& \qquad \mathsf{row\_number}()\ \mathsf{over}\ (\mathsf{order\ by}\ z.\mathsf{i1\_name}, z.\mathsf{i2}, \\
& \qquad\qquad y.\mathsf{dept}, y.\mathsf{name}, y.\mathsf{client})\ \mathsf{as}\ \mathsf{i2\_tasks\_2} \\
& \ \mathsf{from}\ \mathit{contacts}\ \mathsf{as}\ y,\ q\ \mathsf{as}\ z \\
& \ \mathsf{where}\ (z.\mathsf{i1\_name} = y.\mathsf{dept} \land y.\mathsf{client}))
\end{array}
$$

Modulo record flattening, the above fragment of SQL is almost isomorphic to the image of the let-insertion translation. The only significant difference is the use of row_number in place of index. Each instance of index in the body $R$ of a subquery of the form

```
QF1: SELECT e.emp FROM employees e
     WHERE e.salary > 10000
QF2: SELECT e.emp, t.tsk
     FROM employees e, tasks t
     WHERE e.emp = t.emp
QF3: SELECT e1.emp, e2.emp
     FROM employees e1, employees e2
     WHERE e1.dpt = e2.dpt
       AND e1.salary = e2.salary
       AND e1.emp <> e2.emp
QF4: (SELECT t.emp FROM tasks t
      WHERE t.tsk = 'abstract')
     UNION ALL (SELECT e.emp FROM employees
                WHERE e.salary > 50000)
QF5: (SELECT t.emp FROM tasks t
      WHERE t.tsk = 'abstract')
     MINUS
     (SELECT e.emp FROM employees e
      WHERE e.salary > 50000)
QF6: ((SELECT t.emp FROM tasks t
       WHERE t.tsk = 'abstract')
      UNION ALL (SELECT e.emp FROM employees e
                 WHERE e.salary > 50000))
     MINUS
     ((SELECT t.emp FROM tasks t
       WHERE t.tsk = 'enthuse')
      UNION ALL (SELECT e.emp FROM employees e
                 WHERE e.salary > 10000))
```

**Figure 8: SQL versions of flat queries used in experiments**

for $(\overrightarrow{x \leftarrow t}$ where $X)$ return $R$ is simulated by a term of the form row_number() over (order by $\overrightarrow{x.\ell}$), where:

$$x_i : \langle \ell_{i,1} : A_{i,1}, \ldots, \ell_{i,m_i} : A_{i,m_i} \rangle$$
$$\overrightarrow{x.\ell} = x_1.\ell_{1,1}, \ldots, x_1.\ell_{1,m_1}, \quad \ldots, \quad x_n.\ell_{n,1}, \ldots, x_n.\ell_{n,m_n}$$

A possible concern is that row_number is non-deterministic. It computes row numbers ordered by the supplied columns, but if there is a tie, then it is free to order the equivalent rows in any order. However, we always order by *all* columns of all tables referenced from the current subquery, so our use of row_number is always deterministic. (An alternative could be to use nonstandard features such as PostgreSQL's OID or MySQL's ROWNUM, but sorting would still be necessary to ensure consistency across inner and outer queries.)

# 8. EXPERIMENTAL EVALUATION

Ferry's loop-lifting translation has been implemented in Links previously by Ulrich, a member of the Ferry team [30], following the approach described by Grust et al. [12] to generate SQL:1999 algebra plans, then calling Pathfinder [14] to optimise and evaluating the resulting SQL on a PostgreSQL database. We have also implemented query shredding in Links, running against PostgreSQL; our implementation[1] does not use Pathfinder. We performed initial experiments with a larger ad hoc query benchmark, and developed some optimisations, including inlining certain `WITH` clauses to unblock rewrites, using keys for row numbering, and implementing stitching in one pass to avoid construction of intermediate in-memory data structures that are only used once and then discarded. We report on shredding with all of these optimisations enabled.

*Benchmark queries.* There is no standard benchmark for queries returning nested results. In particular, the popular TPC-H benchmark is not suitable for comparing shredding and loop-lifting: the TPC-H queries do not return nested results, and can be run directly on any SQL-compliant RDBMS, so neither approach needs to do any work to produce an SQL query.

[1] http://github.com/slindley/links/tree/shredding

Q1 : for $(d \leftarrow departments)$
return $(\langle \mathrm{name} = d.\mathrm{name},$
$\mathrm{employees} = employeesOfDept\ d,$
$\mathrm{contacts} = contactsOfDept\ d\rangle$

Q2 : for $(d \leftarrow$ Q1$)$
where $(all\ d.\mathrm{employees}\ (\lambda x.contains\ x.\mathrm{tasks}\ \text{“abstract”}))$
return $\langle \mathrm{dept} = d.\mathrm{dept}\rangle$

Q3 : for $(e \leftarrow employees)$
return $\langle \mathrm{name} = e.\mathrm{name}, \mathrm{task} = tasksOfEmp(e)\rangle$

Q4 : for $(d \leftarrow departments)$
return $\langle \mathrm{dept} = d.\mathrm{dept}, \mathrm{employees} =$ for $(e \leftarrow employees)$
where $(d.\mathrm{dept} = e.\mathrm{dept})$
return $e.\mathrm{employee}\rangle$

Q5 : for $(t \leftarrow tasks)$ return $\langle a = t.\mathrm{task}, b = employeesByTask\ t\rangle$

Q6 : for $(d \leftarrow$ Q1$)$
return $(\langle \mathrm{department} = d.\mathrm{name},$
$\mathrm{people} =$
$getTasks(outliers(d.\mathrm{employees}))\ (\lambda y.\ y.\mathrm{tasks})$
$\uplus\ getTasks(clients(d.\mathrm{contacts}))\ (\lambda y.$ return “buy”$)\rangle)$

**Figure 9: Nested queries used in experiments**

We selected twelve queries over the organisation schema described in Section 3 to use as a benchmark. The first six queries, named `QF1`–`QF6`, return flat results and can be translated to SQL using existing techniques, without requiring either shredding or loop-lifting. We considered these queries as a sanity check and in order to measure the overhead introduced by loop-lifting and shredding. Figure 8 shows the SQL versions of these queries.

We also selected six queries `Q1`–`Q6` that do involve nesting, either within query results or in intermediate stages. They are shown in Figure 9; they use the auxiliary functions defined in Section 3. `Q1` is the query $Q_{org}$ that builds the nested organisation view from Section 3. `Q2` is a flat query that computes a flat result from `Q1` consisting of all departments in which all employees can do the “abstract” task; it is a typical example of a query that employs higher-order functions. `Q3` returns records containing each employee and the list of tasks they can perform. `Q4` returns records containing departments and the set of employees in each department. `Q5` returns a record of tasks paired up with sets of employees and their departments. `Q6` is the outliers query $Q$ introduced in Section 3.

*Experimental results.* We measured the query execution time for all queries on randomly generated data, where we vary the number of departments in the organisation from 4 to 4096 (by powers of 2). Each department has on average 100 employees and each employee has 0–2 tasks, and the largest (4096 department) database was approximately 500MB. Although the test datasets are moderate in size, they suffice to illustrate the asymptotic trends in the comparative performance of the different techniques. All tests were performed using PostgreSQL 9.2 running on a MacBook Pro with 4-core 2.6GHz CPU, 8GB RAM and 500GB SSD storage, with the database server running on the same machine (hence, negligible network latency). We measure *total* time to translate a nested query to SQL, evaluate the resulting SQL queries, and stitch the results together to form a nested value, measured from Links.

We evaluated each query using query shredding and loop-lifting, and for the flat queries we also measured the time for Links' default

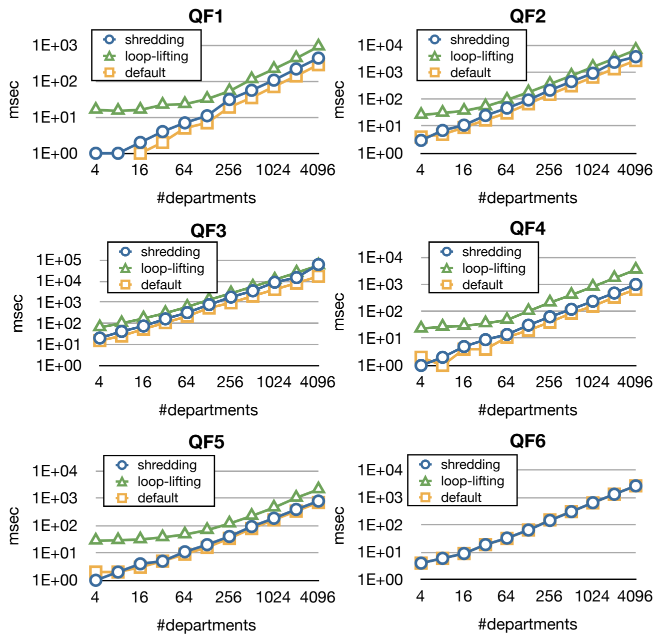


**Figure 10: Experimental results (flat queries)**

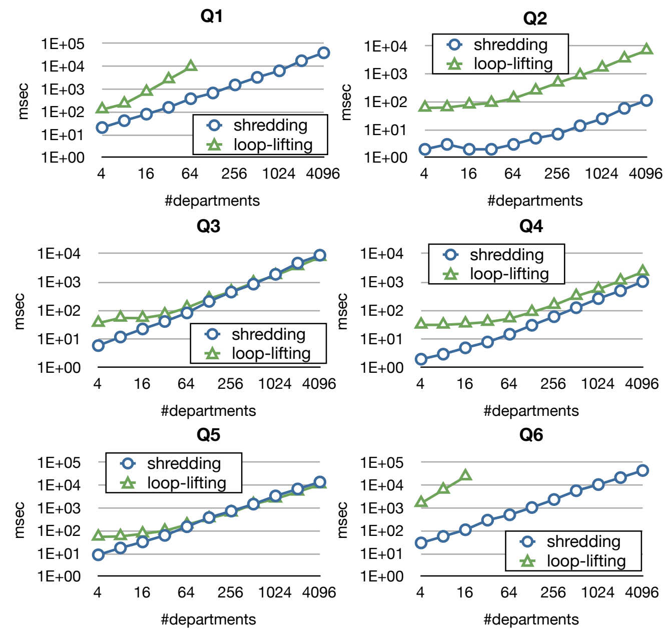


**Figure 11: Experimental results (nested queries)**

(flat) query evaluation. The experimental results for the flat queries are shown in Figure 10 and for the nested queries in Figure 11. Note that both axes are logarithmic. All times are in milliseconds; the times are medians of 5 runs. The times for small data sizes provide a comparison of the overhead associated with shredding, loop-lifting or Links' default query normalisation algorithm.

*Discussion.* We should re-emphasise that Ferry (and Ulrich's loop-lifting implementation for Links) supports grouping and aggregation features that are not handled by our translation. We focused on queries that both systems can handle, but Ferry has a clear advantage for grouping and aggregation queries or when ordering is important (e.g. sorting or top-$k$ queries). Ferry is based on list semantics, while our approach handles multiset semantics. So, some performance differences may be due to our exploitation of multiset-based optimisations that Ferry (by design) does not exploit.

The results for flat queries show that shredding has low per-query overhead in most cases compared to Links' default flat query evaluation, but the queries it generates are slightly slower. Loop-lifting, on the other hand, has a noticeable per-query overhead, likely due to its invocation of Pathfinder and associated serialisation costs. In some cases, such as `QF4` and `QF5`, loop-lifting is noticeably slower asymptotically; this appears to be due to additional sorting needed to maintain list semantics. We encountered a bug that prevented loop-lifting from running on query `QF6`; however, shredding had negligible overhead for this query. In any case, the overhead of either shredding or loop-lifting for flat queries is irrelevant: we can simply evaluate such queries using Links' default flat query evaluator. Nevertheless, these results show that the overhead of shredding for such queries is not large, suggesting that the queries it generates are similar to those currently generated by Links. (Manual inspection of the generated queries confirms this.)

The results for nested queries are more mixed. In most cases, the overhead of loop-lifting is dominant on small data sizes, which again suggests that shredding may be preferable for OLTP or Web workloads involving rapid, small queries. Loop-lifting scales poorly on two queries (`Q1` and `Q6`), and did not finish within 1 minute even for small data sizes. Both `Q1` and `Q6` involve 3 levels of nesting and in the innermost query, loop-lifting generates queries with Cartesian products inside OLAP operators such as `DENSE_RANK` or `ROW_NUMBER` that Pathfinder was not able to remove. The queries generated by shredding in these cases avoid this pathological behaviour. For other queries, such as `Q2` and `Q4`, loop-lifting performs better but is still slower than shredding as data size increases. Comparison of the queries generated by loop-lifting and shredding reveals that loop-lifting encountered similar problems with hard-to-optimise OLAP operations. Finally, for `Q3` and `Q5`, shredding is initially faster (due to the overhead of loop-lifting and calling Pathfinder) but as data size increases, loop-lifting wins out. Inspection of these generated queries reveals that the queries themselves are similar, but the shredded queries involve more data movement. Also, loop-lifting returns sorted results, so it avoids in-memory hashing or sorting while constructing the nested result. It should be possible to incorporate similar optimisations into shredding to obtain comparable performance.

Our experiments show that shredding performs similarly or better than loop-lifting on our (synthetic) benchmark queries on moderate (up to 500MB) databases. Further work may need to be done to investigate scalability to larger databases or consider more realistic query benchmarks.

# 9. RELATED AND FUTURE WORK

We have discussed related work on nested queries, Links, Ferry and LINQ in the introduction. Besides Cooper [7], several authors have recently considered higher-order query languages. Benedikt et al. [2, 32] study higher-order queries over flat relations. The Links approach was adapted to LINQ in F# by Cheney et al. [6]. Higher-order features are also being added to XQuery 3.0 [25].

Research on shredding XML data into relations and evaluating XML queries over such representations [18] is superficially similar to our work in using various indexing or numbering schemes to handle nested data. Grust et al.'s work on translating XQuery to SQL via Pathfinder [14] is a mature solution to this problem, and Grust et al. [13] discuss optimisations in the presence of unordered data processing in XQuery. However, XQuery's ordered data model and treatment of node identity would block transformations in our algorithm that assume unordered, pure operations.

We can now give a more detailed comparison of our approach with the indexing strategies in Van den Bussche's work and in Ferry. Van den Bussche's approach uses natural indexes (that is, $n$-tuples of ids), but does not preserve multiset semantics. Our approach preserves multiplicity and can use natural indexes, we also propose a flat indexing scheme based on row_number. In Ferry's indexing scheme, the surrogate indexes only link adjacent nesting levels, whereas our indexes take information at all higher levels into account. Our flat indexing scheme relies on this property, and Ferry's approach does not seem to be an instance of ours (or vice versa). Ferry can generate multiple SQL:1999 operations and Pathfinder tries to merge them but cannot always do so. Our approach generates row_number operations only at the end, and does not rely on Pathfinder. Finally, our approach uses normalisation and tags parts of the query to disambiguate branches of unions.

Loop-lifting has been implemented in Links by Ulrich [30], and Grust and Ulrich [16] recently presented techniques for supporting higher-order functions as query results. By using Ferry's loop-lifting translation and Pathfinder, Ulrich's system also supports list semantics and aggregation and grouping operations; to our knowledge, it is an open problem to either prove their correctness or adapt these techniques to fit our approach. Ferry's approach supports a list-based semantics for queries, while we assume a bag-based semantics (matching SQL's default behaviour). Either approach can accommodate set-based semantics simply by eliminating duplicates in the final result. In fact, however, we believe the core query shredding translation (Sections 4–6) works just as well for a list semantics. The only parts that rely on unordered semantics are normalisation (Section 2.2) and conversion to SQL (Section 7). We leave these extensions to future work.

Our work is also partly inspired by work on unnesting for nested data parallelism. Blelloch and Sabot [3] give a compilation scheme for NESL, a data-parallel language with nested lists; Suciu and Tannen [27] give an alternative scheme for a nested list calculus. This work may provide an alternative (and parallelisable) implementation strategy for Ferry's list-based semantics [12].

# 10. CONCLUSION

Combining efficient database access with high-level programming abstractions is challenging in part because of the limitations of flat database queries. Currently, programmers must write flat queries and manually convert the results to a nested form. This damages declarativity and maintainability. Query shredding can help to bridge this gap. Although it is known from prior work that query shredding is possible in principle, and some implementations (such as Ferry) support this, getting the details right is tricky, and can lead to queries that are not optimised well by the relational engine. Our contribution is an alternative shredding translation that handles queries over bags and delays use of OLAP operations until the final stage. Our translation compares favourably to loop-lifting in performance, and should be straightforward to extend and to incorporate into other language-integrated query systems.

*Acknowledgements.* We thank Ezra Cooper, Torsten Grust, and Jan Van den Bussche for comments. We are very grateful to Alex Ulrich for sharing his implementation of Ferry in Links and for extensive discussions. This work was supported by EPSRC grants EP/J014591/1 and EP/K034413/1 and a Google Research Award.

# 11. REFERENCES

[1] A. Beckmann. Exact bounds for lengths of reductions in typed $\lambda$-calculus. *Journal of Symbolic Logic*, 66:1277–1285, 2001.
[2] M. Benedikt, G. Puppis, and H. Vu. Positive higher-order queries. In *PODS*, pages 27–38, 2010.
[3] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8:119–134, February 1990.
[4] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23, 1994.
[5] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.
[6] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *ICFP*, pages 403–416. ACM, 2013.
[7] E. Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL*, pages 36–51, 2009.
[8] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *FMCO*, volume 4709 of *LNCS*, 2007.
[9] P. de Groote. On the strong normalisation of intuitionistic natural deduction with permutation-conversions. *Inf. Comput.*, 178(2):441–464, 2002.
[10] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25:457–516, December 2000.
[11] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: Database-supported program execution. In *SIGMOD*, June 2009.
[12] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe LINQ compilation. *PVLDB*, 3(1), 2010.
[13] T. Grust, J. Rittinger, and J. Teubner. eXrQuy: Order indifference in XQuery. In *ICDE*, pages 226–235, 2007.
[14] T. Grust, J. Rittinger, and J. Teubner. Pathfinder: XQuery off the relational shelf. *IEEE Data Eng. Bull.*, 31(4):7–14, 2008.
[15] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL hosts. In *VLDB*, pages 252–263, 2004.
[16] T. Grust and A. Ulrich. First-class functions for first-order database engines. In *DBPL*, 2013.
[17] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, Sept. 1982.
[18] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Xsym*, pages 1–18, 2003.
[19] S. Lindley. Extensional rewriting with sums. In *TLCA*, 2007.
[20] S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI*, 2012.
[21] S. Lindley and I. Stark. Reducibility and $\top\top$-lifting for computation types. In *TLCA*, 2005.
[22] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.
[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.
[24] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.
[25] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML query language. W3C Proposed Recommendation, October 2013. http://www.w3.org/TR/xquery-30/.
[26] H. J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11:137–147, April 1986.
[27] D. Suciu and V. Tannen. Efficient compilation of high-level data parallel algorithms. In *SPAA*, pages 57–66. ACM, 1994.
[28] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. Apress, 2012.
[29] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.
[30] A. Ulrich. A Ferry-based query backend for the Links programming language. Master's thesis, University of Tübingen, 2011. Code supplement: http://github.com/slindley/links/tree/sand.
[31] J. Van den Bussche. Simulation of the nested relational algebra by

the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theor. Comput. Sci.*, 254(1-2), 2001.

[32] H. Vu and M. Benedikt. Complexity of higher-order queries. In *ICDT*, pages 208–219. ACM, 2011.

[33] P. Wadler. Comprehending monads. *Math. Struct. in Comp. Sci.*, 2(4), 1992.

[34] L. Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3), 1996.

[35] L. Wong. Kleisli, a functional query system. *J. Funct. Programming*, 10(1), 2000.

# APPENDIX

## A. BEHAVIOUR OF VAN DEN BUSSCHE'S SIMULATION ON MULTISETS

As noted in the Introduction, Van den Bussche's simulation of nested queries via flat queries does not work properly over multisets. To illustrate the problem, consider a simple query $R \cup S$, where $R$ and $S$ have the same schema Bag $\langle A : Int, B : \text{Bag } Int\rangle$. Suppose $R$ and $S$ have the following values:

$R =$

| $A$ | $B$ |
|---|---|
| 1 | $\{1\}$ |
| 2 | $\{2\}$ |

$S =$

| $A$ | $B$ |
|---|---|
| 1 | $\{3,4\}$ |
| 2 | $\{2\}$ |

then their multiset union is:

$R \cup S =$

| $A$ | $B$ |
|---|---|
| 1 | $\{1\}$ |
| 1 | $\{3,4\}$ |
| 2 | $\{2\}$ |
| 2 | $\{2\}$ |

Van den Bussche's simulation (like ours) represents these nested values by queries over flat tables, such as the following:

$R_1 =$

| $A$ | $id$ |
|---|---|
| 1 | $a$ |
| 2 | $b$ |

$R_2 =$

| $id$ | $B$ |
|---|---|
| $a$ | 1 |
| $b$ | 2 |

$S_1 =$

| $A$ | $id$ |
|---|---|
| 1 | $a$ |
| 2 | $b$ |

$S_2 =$

| $id$ | $B$ |
|---|---|
| $a$ | 3 |
| $a$ | 4 |
| $b$ | 2 |

where $a, b$ are arbitrary distinct ids. Note however that $R$ and $S$ have overlapping ids, so if we simply take the union of $R_1$ and $S_1$, and of $R_2$ and $S_2$ respectively, we will get:

$Wrong_1 =$

| $A$ | $id$ |
|---|---|
| 1 | $a$ |
| 2 | $b$ |
| 1 | $a$ |
| 2 | $b$ |

$Wrong_2 =$

| $id$ | $B$ |
|---|---|
| $a$ | 1 |
| $b$ | 2 |
| $a$ | 3 |
| $a$ | 4 |
| $b$ | 2 |

corresponding to nested value:

$Wrong =$

| $A$ | $B$ |
|---|---|
| 1 | $\{1,3,4\}$ |
| 1 | $\{1,3,4\}$ |
| 2 | $\{2,2\}$ |
| 2 | $\{2,2\}$ |

Instead, both Van den Bussche's simulation and our approach avoid clashes among ids when taking unions. Van den Bussche's simulation does this by adding two new id fields to represent the result of a union, say $id_1$ and $id_2$. The tuples originating from $R$ will have *equal* $id_1$ and $id_2$ values, while those originating from $S$ will have *different* $id_1$ and $id_2$ values. In order to do this, the simulation defines two queries, one for each result table. The first query is of the form:

$$\begin{aligned} T_1 &= (R_1 \times (id_1 : x, id_2 : x) \mid x \in \textbf{adom}) \\ &\cup (S_1 \times (id_1 : x, id_2 : x') \mid x \neq x' \in \textbf{adom}) \end{aligned}$$

and similarly

$$\begin{aligned} T_2 &= (R_2 \times (id_1 : x, id_2 : x) \mid x \in \textbf{adom}) \\ &\cup (S_2 \times (id_1 : x, id_2 : x') \mid x \neq x' \in \textbf{adom}) \end{aligned}$$

where **adom** is the active domain of the database (in this case, **adom** $= \{1, 2, 3, 4, a, b\}$) — this is of course also definable as a query. Thus, in this example, the result is of the form

$T_1 =$

| $A$ | $id$ | $id_1$ | $id_2$ |
|---|---|---|---|
| 1 | $a$ | $x$ | $x$ |
| 2 | $b$ | $y$ | $y$ |
| 1 | $a$ | $z$ | $z'$ |
| 2 | $b$ | $v$ | $v'$ |

$T_1 =$

| $id$ | $id_1$ | $id_2$ | $B$ |
|---|---|---|---|
| $a$ | $x$ | $x$ | 1 |
| $b$ | $y$ | $y$ | 2 |
| $a$ | $z$ | $z'$ | 3 |
| $a$ | $w$ | $w'$ | 4 |
| $b$ | $v$ | $v'$ | 2 |

where $x, y$ are any elements of **adom** (rows mentioning $x$ and $y$ stand for 6 instances) and $z \neq z'$, $w \neq w'$ and $v \neq v'$ are any pairs of distinct elements of **adom** (rows mentioning these variables stand for 30 instances of distinct pairs from **adom**). This leads to an $O(|\mathbf{adom}| * |R| + |\mathbf{adom}|^2 * |S|)$ blowup in the number of tuples. Specifically, for our example, $|T_1| = 72$, whereas the actual number of tuples in a natural representation of $R \cup S$ is only 9. In a set semantics, the set value simulated by these tables is correct even with all of the extra tuples; however, for a multiset semantics, this quadratic blowup is not correct — even for our example, with $|\mathbf{adom}| = 6$, the result of evaluating $R \cup S$ yields a different number of tuples from the result of evaluating $S \cup R$, and neither represents the correct multiset in an obvious way. It may be possible (given knowledge of the query and active domain, but not the source database) to "decode" the flat query results and obtain the correct nested result, but doing so appears no easier than developing an alternative, direct algorithm.

# B. TYPING RULES

The (standard) typing rules for $\lambda_{NRC}$ queries are shown in Figure 12. The typing rules for shredded terms, packages, and values are shown in Figures 13, 14, 16 and 18.

# C. QUERY NORMALISATION

Following previous work [20], we separate query normalisation into a rewriting phase and a type-directed structurally recursive function. Where we diverge from this previous work is that we extend the rewrite relation to hoist all conditionals up to the nearest comprehension (in order to simplify the rest of the development), and the structurally recursive function is generalised to handle nested data. Thus normalisation can be divided into three stages.

- The first stage performs symbolic evaluation, that is, $\beta$-reduction and commuting conversions, flattening unnecessary nesting and eliminating higher-order functions.
- The second stage hoists all conditionals up to the nearest enclosing comprehension in order to allow them to be converted to where clauses.
- The third and final stage hoists all unions up to the top-level, $\eta$-expands tables and variables, and turns all conditionals into where clauses.

*Weak normalisation and strong normalisation.* Given a term $M$ and a rewrite relation $\leadsto$, we write $M \not\leadsto$ if $M$ is irreducible, that is no rewrite rules in $\leadsto$ apply to $M$. The term $M$ is said to be in *normal form*.

A term $M$ is *weakly normalising* with respect to a rewrite relation $\leadsto_r$, or $r$-*WN*, if there exists a finite reduction sequence

$$M \leadsto M_1 \leadsto \ldots \leadsto M_n \not\leadsto$$

A term $M$ is *strongly normalising* with respect to a rewrite relation $\leadsto_r$, or $r$-*SN*, if every reduction sequence starting from $M$ is finite. If $M$ is $r$-SN, then we write $max_r(M)$ for the maximum length of a reduction sequence starting from $M$.

A rewrite relation $\leadsto_r$ is *weakly-normalising*, or *WN*, if all terms $M$ are weakly-normalising with respect to $\leadsto_r$. Similarly, a rewrite relation $\leadsto_r$ is *strongly-normalising*, or *SN*, if all terms $M$ are strongly-normalising with respect to $\leadsto_r$.

For any $\leadsto_r$ we can define a partial function $nf_r$ such that $nf_r(M)$ is a normal form of $M$, if one exists, otherwise $nf_r(M)$ is undefined. Note that there may be more than one normal form for a given $M$; $nf_r$ chooses a particular normal form for each $M$ that has one. If $\leadsto_r$ is weakly normalising, then $nf_r$ is a total function.

We now describe each normalisation stage in turn.

## C.1 Symbolic evaluation

The $\beta$-rules perform symbolic evaluation, including substituting argument values for function parameters, record field projection, conditionals where the test is known to be true or false, or iteration over singleton bags.

$$\begin{array}{rcl} (\lambda x.\, N)\ M & \leadsto_c & N[x := M] \\ \langle\overrightarrow{\ell = M}\rangle.\ell_i & \leadsto_c & M_i \\ \mathsf{if\ true\ then}\ M\ \mathsf{else}\ N & \leadsto_c & M \\ \mathsf{if\ false\ then}\ M\ \mathsf{else}\ N & \leadsto_c & N \\ \mathsf{for}\ (x \leftarrow \mathsf{return}\ M)\ N & \leadsto_c & N[x := M] \end{array}$$

Fundamentally, $\beta$-rules always follow the same pattern. Each is associated with a particular type constructor $T$, and the left-hand side always consists of an *introduction* form for $T$ inside an *elimination* form for $T$. For instance, in the case of functions (the first $\beta$-rule above), the introduction form is a lambda and the elimination form is an application. Applying a $\beta$-rule *eliminates* $T$ in the sense that the introduction form from the left-hand side either no longer appears or has been replaced by a term of a simpler type on the right-hand side.

Each instance of a $\beta$-rule is associated with an *elimination frame*. An *elimination frame* is simply the elimination form with a designated *hole* $[\ ]$ that can be *plugged* with another expression. For instance, elimination frames for the function $\beta$-rule are of the form $E[\ ] = [\ ]\ M$. If we plug an introduction form $\lambda x.N$ into $E[\ ]$, written $E[\lambda x.N]$, then we obtain the left-hand side of the associated $\beta$-rule $(\lambda x.N)\ M$.

(This notion of an expression with a hole that can be filled by another expression is commonly used in rewriting and operational semantics for programming languages [24, ch. 19]; here, it is not essential but helps cut down the number of explicit rules, and helps highlight commonality between rules.) The elimination frames of $\lambda_{NRC}$ are as follows.

$$E[\,] ::= [\,]\ M \mid [\,].\ell \mid \mathsf{if}\ [\,]\ \mathsf{then}\ M\ \mathsf{else}\ N \mid \mathsf{for}\ (x \leftarrow [\,])\ N$$

The following rules express that comprehensions, conditionals, empty bag constructors, and unions can always be *hoisted* out of the above elimination frames. In the literature such rules are often called *commuting conversions*. They are necessary in order to expose all possible $\beta$-reductions. For instance,

$$(\mathsf{if}\ M\ \mathsf{then}\ \langle\ell = N\rangle\ \mathsf{else}\ N').\ell$$

cannot $\beta$-reduce, but $\mathsf{if}\ M\ \mathsf{then}\ \langle\ell = N\rangle.\ell\ \mathsf{else}\ N'.\ell$ can.

$$\begin{array}{rcl} E[\mathsf{for}\ (x \leftarrow M)\ N] & \leadsto_c & \mathsf{for}\ (x \leftarrow M)\ E[N] \\ E[\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N] & \leadsto_c & \mathsf{if}\ L\ \mathsf{then}\ E[M]\ \mathsf{else}\ E[N] \\ E[\emptyset] & \leadsto_c & \emptyset \\ E[M_1 \uplus M_2] & \leadsto_c & E[M_1] \uplus E[M_2] \end{array}$$

For example:

$$(\mathsf{if}\ L\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)\ M \leadsto_c \mathsf{if}\ L\ \mathsf{then}\ M_1\ M\ \mathsf{else}\ M_2\ M$$

$$\textsc{Var}\quad \frac{}{\Gamma, x : A \vdash x : A} \qquad \textsc{Constant}\quad \frac{\Sigma(c) = \langle O_1, \ldots, O_n \rangle \to O' \qquad [\Gamma \vdash M_i : O_i]_{i=1}^n}{\Gamma \vdash c(M_1, \ldots, M_n) : O'} \qquad \textsc{Lam}\quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B}$$

$$\textsc{App}\quad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B} \qquad \textsc{Record}\quad \frac{[\Gamma \vdash M_i : A_i]_{i=1}^n}{\Gamma \vdash \langle \ell_i = M_i \rangle_{i=1}^n : \langle \ell_i = A_i \rangle_{i=1}^n} \qquad \textsc{Project}\quad \frac{\Gamma \vdash M : \langle \ell_i : A_i \rangle_{i=1}^n}{\Gamma \vdash M.\ell_j : A_j}$$

$$\textsc{If}\quad \frac{\Gamma \vdash M : \mathit{Bool} \qquad \Gamma \vdash N : A \qquad \Gamma \vdash N' : A}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ N' : A} \qquad \textsc{Empty}\quad \frac{}{\Gamma \vdash \emptyset : \mathrm{Bag}\,A} \qquad \textsc{Singleton}\quad \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{return}\ M : \mathrm{Bag}\,A} \qquad \textsc{Union}\quad \frac{\Gamma \vdash M : \mathrm{Bag}\,A \qquad \Gamma \vdash N : \mathrm{Bag}\,A}{\Gamma \vdash M \uplus N : \mathrm{Bag}\,A}$$

$$\textsc{For}\quad \frac{\Gamma \vdash M : \mathrm{Bag}\,A \qquad \Gamma, x : A \vdash N : \mathrm{Bag}\,B}{\Gamma \vdash \mathsf{for}\,(x \leftarrow M)\,N : \mathrm{Bag}\,B} \qquad \textsc{IsEmpty}\quad \frac{\Gamma \vdash M : \mathrm{Bag}\,A}{\Gamma \vdash \mathsf{empty}\ M : \mathit{Bool}} \qquad \textsc{Table}\quad \frac{\Sigma(t) = \mathrm{Bag}\,\langle \overrightarrow{\ell : O} \rangle}{\Gamma \vdash \mathsf{table}\ t : \mathrm{Bag}\,\langle \overrightarrow{\ell : O} \rangle}$$

**Figure 12: Typing rules for higher-order nested queries**

$$\textsc{Var}\quad \frac{}{\Gamma, x : \langle \overrightarrow{\ell : F} \rangle \vdash x : \langle \overrightarrow{\ell : F} \rangle} \qquad \textsc{Constant}\quad \frac{\Sigma(c) = \langle O_1, \ldots, O_n \rangle \to O' \qquad [\Gamma \vdash X_i : O_i]_{i=1}^n}{\Gamma \vdash c(X_1, \ldots, X_n) : O'} \qquad \textsc{Record}\quad \frac{[\Gamma \vdash N_i : A_i]_{i=1}^n}{\Gamma \vdash \langle \ell_i = N_i \rangle_{i=1}^n : \langle \ell_i = A_i \rangle_{i=1}^n}$$

$$\textsc{Project}\quad \frac{\Gamma \vdash x : \langle \ell_i : A_i \rangle_{i=1}^n}{\Gamma \vdash x.\ell_j : A_j} \qquad \textsc{Index}\quad \frac{}{\Gamma \vdash a \diamond d : \mathit{Index}} \qquad \textsc{Singleton}\quad \frac{\Gamma \vdash I : \mathit{Index} \qquad \Gamma \vdash N : F}{\Gamma \vdash \mathsf{return}^a\ \langle I, N \rangle : \mathrm{Bag}\,\langle \mathit{Index}, F \rangle} \qquad \textsc{Union}\quad \frac{[\Gamma \vdash C_i : \mathrm{Bag}\,\langle \mathit{Index}, F \rangle]_{i=1}^n}{\Gamma \vdash \biguplus \vec{C} : \mathrm{Bag}\,\langle \mathit{Index}, F \rangle}$$

$$\textsc{For}\quad \frac{[\Sigma(t_i) = \mathrm{Bag}\,A_i]_{i=1}^n \qquad \Gamma, [x_i : A_i]_{i=1}^n \vdash X : \mathit{Bool} \qquad \Gamma, [x_i : A_i]_{i=1}^n \vdash C : \mathrm{Bag}\,\langle \mathit{Index}, F \rangle}{\Gamma \vdash \mathsf{for}\,([x_i \leftarrow t_i]_{i=1}^n\ \mathsf{where}\ X)\,C : \mathrm{Bag}\,\langle \mathit{Index}, F \rangle} \qquad \textsc{IsEmpty}\quad \frac{\Gamma \vdash L : \mathrm{Bag}\,\langle \mathit{Index}, F \rangle}{\Gamma \vdash \mathsf{empty}\ L : \mathit{Bool}}$$

**Figure 13: Typing rules for shredded terms**

$$\textsc{Base}\quad \frac{}{\vdash O(\Lambda_S) : O(\mathcal{T}_S)} \qquad \textsc{Record}\quad \frac{[\vdash \hat{A}_i(\Lambda_S) : \hat{A}'_i(\mathcal{T}_S)]_{i=1}^n}{\vdash \langle \ell_i : \hat{A}_i(\Lambda_S) \rangle_{i=1}^n : \langle \ell_i : \hat{A}'_i(\mathcal{T}_S) \rangle_{i=1}^n} \qquad \textsc{Bag}\quad \frac{\vdash \hat{A}(\Lambda_S) : \hat{A}'(\mathcal{T}_S) \qquad \vdash L : A \qquad \mathit{erase}(\hat{A}(\Lambda_S)) = A}{\vdash (\mathrm{Bag}\,\hat{A}(\Lambda_S))^L : (\mathrm{Bag}\,\hat{A}'(\mathcal{T}_S))^A}$$

We write $\Lambda_S$ for the set of shredded terms, and $\mathcal{T}_S$ for the set of shredded types.

**Figure 14: Typing rules for shredded packages**

Note that some combinations of elimination frames and rewrite rule are impossible in a well-typed term, such as if $\emptyset$ then $M$ else $N$. For the purposes of reduction we treat empty like an uninterpreted constant, that is, we do reduce inside emptiness tests, but they do not in any other way interact with the reduction rules.

Next we prove that $\rightsquigarrow_c$ is strongly normalising. The proof is based on a previous proof of strong normalisation for simply-typed $\lambda$-calculus with sums [19], which generalises the $\top\top$-lifting approach [21], which in turn extends Tait's proof of strong normalisation for simply-typed $\lambda$- calculus [29].

### *Frame stacks.*

$$\begin{array}{lrcl}
\text{(frame stacks)} & S & ::= & Id \mid S \circ E \\
\text{(stack length)} & |Id| & = & 0 \\
 & |S \circ E| & = & |S| + 1 \\
\text{(plugging)} & Id[M] & = & M \\
 & (S \circ E)[M] & = & S[(E[M])]
\end{array}$$

Following previous work [19] we assume variables are annotated with types. We assume variables are annotated with types. We write $A \multimap B$ for the type of frame stack $S$, if $S[M] : B$ for all terms $M : A$.

### *Frame stack reduction.*

$$\begin{array}{lcl}
S \rightsquigarrow_c S' & \overset{def}{\iff} & \forall M. S[M] \rightsquigarrow_c S'[M] \\
 & \iff & S[x] \rightsquigarrow_c S'[x]
\end{array}$$

Frame stacks are closed under reduction. A frame stack $S$ is *c-strongly normalising*, or *c-SN*, if all reduction sequences starting from $S$ are finite.

LEMMA 7. *If $S \rightsquigarrow_c S'$, for frame stacks $S, S'$, then $|S'| \leq |S|$.*

PROOF. Induction on the structure of $S$. $\square$

***Reducibility.*** We define *reducibility* as follows:

- $Id$ is reducible.
- $S \circ ([\,]\ N) : (A \to B) \multimap C$ is reducible if $S$ and $N$ are reducible.
- $S \circ ([\,].\ell) : (\overrightarrow{\ell : A}) \multimap C$ is reducible if $S$ is reducible.
- $S : \mathrm{Bag}\, A \multimap C$ is reducible if $S[\mathsf{return}\, M]$ is $c$-SN for all reducible $M : A$.
- $S : Bool \multimap C$ is reducible if $S[\mathsf{true}]$ is $c$-SN and $S[\mathsf{false}]$ is $c$-SN.
- $M : A$ is reducible if $S[M]$ is $c$-SN for all reducible $S : A \multimap C$.

LEMMA 8. *If $M : A$ is reducible then $M$ is c-SN.*

PROOF. Follows immediately from reducibility of $Id$ and the definition of reducibility on terms. $\square$

LEMMA 9. *$x : A$ is reducible.*

PROOF. By induction on $A$ using Lemma 7 and Lemma 8. $\square$

COROLLARY 10. *If $S : A \multimap C$ is reducible then $S$ is c-SN.*

Each type constructor has an associated $\beta$-rule. Each $\beta$-rule gives rise to an SN-closure property.

LEMMA 11 (SN-CLOSURE).

($\to$) *If $S[M[x := N]]$ and $N$ are c-SN then $S[(\lambda x.M)\ N]$ is c-SN.*

($\langle\rangle$) *If $\vec{M}$, are c-SN then $S[\langle\overrightarrow{\ell = M}\rangle.\ell_i]$ is c-SN.*

(Bag $-$) *If $S[N[x := M]]$ and $M$ are c-SN then $S[\mathsf{for}\ (x \leftarrow \mathsf{return}\ M)\ N]$ is c-SN.*

($Bool$) *If $S[N]$ and $S[N']$ are c-SN then $S[\mathsf{if\ true\ then}\ N\ \mathsf{else}\ N']$ is c-SN and $S[\mathsf{if\ false\ then}\ N\ \mathsf{else}\ N']$ is c-SN.*

PROOF.

**($\to$):** By induction on $\max_c(S) + \max_c(M) + \max_c(N)$.

**($\langle\rangle$):** By induction on

$$\max_c S + (\sum_{i=1}^{n} \max_c(M_i)) + \max_c(N) + (\sum_{i=1}^{n} \max_c(M'_i)).$$

**(Bag ):** By induction on

$$|S| + \max_c(S[N[x := M]]) + \max_c(M).$$

**($Bool$):** By induction on $|S| + \max_c(S[N]) + \max_c(S[N'])$.

This completes the proof. $\square$

Now we obtain reducibility-closure properties for each type constructor.

LEMMA 12 (REDUCIBILITY-CLOSURE).

($\to$) *If $M[x := N]$ is reducible for all reducible $N$, then $\lambda x.M$ is reducible.*

($\langle\rangle$) *If $\vec{M}$ are reducible, then $\langle\overrightarrow{\ell = M}\rangle$ is reducible.*

(Bag ) *If $M$ is reducible, $N[x := M']$ is reducible for all reducible $M'$, then $\mathsf{for}\ (x \leftarrow M)\ N$ is reducible.*

($Bool$) *If $M, N, N'$ are reducible then $\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else},\ N'$ is reducible.*

PROOF. Each property follows from the corresponding part of Lemma 11 using Lemma 8 and Corollary 10. $\square$

We also require additional closure properties for the empty bag and union constructs.

LEMMA 13 (REDUCIBILITY-CLOSURE II).

($\emptyset$) *The empty bag $\emptyset$ is reducible.*

($\uplus$) *If $M, N$ are reducible, then $M \uplus N$ is reducible.*

PROOF.

**($\emptyset$):** Suppose $S : \mathrm{Bag}\, A \multimap C$ is reducible. We need to prove that $S[\emptyset]$ is $c$-SN. The proof is by induction on $|S| + \max_c(S)$. The only interesting case is hoisting the empty bag out of a bag elimination frame, which simply decreases the size of the frame stack by 1.

**($\uplus$):** Suppose $M, N : \mathrm{Bag}\, A$, and $S : \mathrm{Bag}\, A \multimap C$ are reducible. We need to show that $S[M \uplus N]$ is $c$-SN. The proof is by induction on $|S| + \max_c(S[M]) + \max_c(S[N])$. The only interesting case is hoisting the union out of a bag elimination frame, which again decreases the size of the frame stack by 1, whilst leaving the other components of the induction measure unchanged.

This completes the proof. $\square$

THEOREM 14. *Let $M$ be any term. Suppose $x_1 : A_1, \ldots, x_n : A_n$ includes all the free variables of $M$. If $N_1 : A_1, \ldots, N_n : A_n$ are reducible then $M[\vec{x} := \vec{N}]$ is reducible.*

PROOF. By induction on the structure of terms using Lemma 12 and Lemma 13. $\square$

THEOREM 15 (STRONG NORMALISATION). *The relation $\leadsto_c$ is strongly normalising.*

PROOF. Let $M$ be a term with free variables $\vec{x}$. By Lemma 9, $\vec{x}$ are reducible. Hence, by Theorem 14, $M$ is $c$-SN. $\square$

It is well known that $\beta$-reduction in simply-typed $\lambda$-calculus has non-elementary complexity in the worst case [1]. The relation $\leadsto_c$ includes $\beta$-reduction, so it must be at least as bad (we conjecture that it has the same asymptotic complexity, as $\leadsto_c$ can be reduced to $\beta$-reduction on simply-typed $\lambda$-calculus via a CPS translation following de Groote [9]). However, we believe that the asymptotic complexity is unlikely to pose a problem in practice, as the kind of higher-order code that exhibits worst-case behaviour is rare. It has been our experience with Links that query normalisation time is almost always dominated by SQL execution time.

## C.2 If hoisting

To hoist conditionals (if-expressions) out of constant applications, records, unions, and singleton bag constructors, we define if-hoisting frames as follows:

$$\begin{aligned} F[\,] ::= {} & c(\vec{M}, [\,], \vec{N}) \mid \langle \overrightarrow{\ell' = M}, \ell = [\,], \overrightarrow{\ell'' = N} \rangle \\ \mid {} & [\,] \uplus N \mid M \uplus [\,] \mid \mathsf{return}\,[\,] \end{aligned}$$

The if-hoisting rule says that if an expression contains an if-hoisting frame around a conditional, then we can lift the conditional up and push the frame into both branches:

$$F[\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N] \leadsto_h \mathsf{if}\ L\ \mathsf{then}\ F[M]\ \mathsf{else}\ F[N]$$

We write $size(M)$ for the size of $M$, as in the total number of syntax constructors in $M$.

LEMMA 16.
1. *If $M, N, N'$ are h-SN then* $\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ N'$ *is h-SN.*
2. *If $M, N$ are h-SN then $M \uplus N$ is h-SN.*
3. *If $\vec{M}$ are h-SN then $c(\vec{M})$ is h-SN.*
4. *If $\vec{M}$ are h-SN then $\langle \overrightarrow{\ell = M} \rangle$ is h-SN.*

PROOF.

**1:** By induction on $\langle max_h(M),\ size(M),\ max_h(N) + max_h(N'), size(N) + size(N') \rangle$.

**2:** By induction on $\langle max_h(M) + max_h(N), size_h(M) + size_h(N) \rangle$ using (1).

**3 and 4:** By induction on

$$\langle \sum_{i=1}^{n} max_h(M_i), \sum_{i=1}^{n} size(M_i) \rangle$$

using (1).

This concludes the proof. $\square$

PROPOSITION 17. *The relation $\leadsto_h$ is strongly normalising.*

PROOF. By induction on the structure of terms using Lemma 16. $\square$

## C.3 The query normalisation function

The following definition of the function $norm$ generalises the normalisation algorithm from previous work [20].

$$norm_A(M) = \llparenthesis nf_h(nf_c(M)) \rrparenthesis_A$$

where:

$$\begin{aligned} \llparenthesis c([X_i : O_i]_{i=1}^n) \rrparenthesis_O &= c([\llparenthesis X_i \rrparenthesis_{O_i}]_{i=1}^n) \\ \llparenthesis x.\ell \rrparenthesis_O &= x.\ell \\ \llparenthesis \mathsf{empty}\ (M : \mathrm{Bag}\ A) \rrparenthesis_{Bool} &= \mathsf{empty}\ (\llparenthesis M \rrparenthesis_{\mathrm{Bag}\ A}) \\ \llparenthesis M \rrparenthesis_{\langle \ell_i : A_i \rangle_{i=1}^n} &= \langle \ell_i = \mathcal{F}\llparenthesis M \rrparenthesis_{A_i, \ell_i} \rangle_{i=1}^n \\ \llparenthesis M \rrparenthesis_{\mathrm{Bag}\ A} &= \biguplus (\mathcal{B}\llparenthesis M \rrparenthesis^\star_{A,[\,],\mathsf{true}}) \end{aligned}$$

$$\begin{aligned} \mathcal{B}\llparenthesis \mathsf{return}\ M \rrparenthesis^\star_{A,\vec{G},L} &= [\mathsf{for}\ (\vec{G}\ \mathsf{where}\ L)\ \mathsf{return}\ \llparenthesis M \rrparenthesis_A] \\ \mathcal{B}\llparenthesis \mathsf{for}\ (x \leftarrow t)\ M \rrparenthesis^\star_{A,\vec{G},L} &= \mathcal{B}\llparenthesis M \rrparenthesis^\star_{A,\vec{G} +\!\!+ [x \leftarrow t],L} \\ \mathcal{B}\llparenthesis \mathsf{table}\ t \rrparenthesis^\star_{A,\vec{G},L} &= \mathcal{B}\llparenthesis \mathsf{return}\ x \rrparenthesis^\star_{A,\vec{G} +\!\!+ [x \leftarrow t],L} \\ & \quad (x \text{ fresh}) \\ \mathcal{B}\llparenthesis \emptyset \rrparenthesis^\star_{A,\vec{G},L} &= [\,] \\ \mathcal{B}\llparenthesis M \uplus N \rrparenthesis^\star_{A,\vec{G},L} &= \mathcal{B}\llparenthesis M \rrparenthesis^\star_{A,\vec{G},L} +\!\!+ \mathcal{B}\llparenthesis N \rrparenthesis^\star_{A,\vec{G},L} \\ \mathcal{B}\llparenthesis \mathsf{if}\ L'\ M\ \ N \rrparenthesis^\star_{A,\vec{G},L} &= \mathcal{B}\llparenthesis M \rrparenthesis^\star_{A,\vec{G},L \wedge L'} \\ & \quad +\!\!+ \mathcal{B}\llparenthesis N \rrparenthesis^\star_{A,\vec{G},L \wedge \neg L'} \end{aligned}$$

$$\begin{aligned} \mathcal{F}\llparenthesis x \rrparenthesis_{A,\ell_i} &= \llparenthesis x.\ell_i \rrparenthesis_A \\ \mathcal{F}\llparenthesis \langle \ell_i = M_i \rangle_{i=1}^n \rrparenthesis_{A,\ell_i} &= \llparenthesis M_i \rrparenthesis_A \end{aligned}$$

Strictly speaking, in order for the above definition of $norm_A$ to make sense we need the two rewrite relations to be confluent. It is easily verified that the relation $\leadsto_c$ is locally confluent, and hence by strong normalisation and Newman's Lemma it is confluent. The relation $\leadsto_h$ is not confluent as the ordering of hoisting determines the final order in which booleans are eliminated. However, it is easily seen to be confluent modulo reordering of conditionals, which in turn means that $norm_A$ is well-defined if we identify terms modulo commutativity of conjunction, which is perfectly reasonable given that conjunction is indeed commutative.

THEOREM 18. *The function $norm_A$ terminates.*

PROOF. The result follows immediately from strong normalisation of $\leadsto_c$ and $\leadsto_h$, and the fact that the functions $\llparenthesis - \rrparenthesis$, $\mathcal{B}\llparenthesis - \rrparenthesis^\star$, and $\mathcal{F}\llparenthesis - \rrparenthesis$ are structurally recursive (modulo expanding out the right-hand-side of the definition of $\mathcal{B}\llparenthesis \mathsf{table}\ t \rrparenthesis^\star_{A,\vec{G},L}$. $\square$

# D. PROOF OF CORRECTNESS OF SHREDDING

The main correctness result is that if we shred an annotated normalised nested query, run all the resulting shredded queries, and stitch the shredded results back together, then that is the same as running the nested query directly.

Intuitively, the idea is as follows:
1. define shredding on nested values;
2. show that shredding commutes with query execution; and
3. show that stitching is a left-inverse to shredding of values.

Unfortunately, this strategy is a little too naive to work, because nested values do not contain enough information about the structure of the source query in order to generate the same indexes that are generated by shredded queries.

To fix the problem, we will give an annotated semantics $\mathcal{A}[\![-]\!]$ that is defined only on queries that have first been converted to normal form and annotated with static indexes as described in Section 4. We will ensure that $\mathcal{N}[\![erase(L)]\!] = erase(\mathcal{A}[\![L]\!])$, where we overload the $erase$ function to erase annotations on terms and values. The structure of the resulting proof is depicted in the diagram in Figure 15, where we view a query as a function from the in-

$$\frac{[\vdash v_i : A_i]_{i=1}^n \qquad [\vdash J : Index]_{i=1}^n}{\vdash [v_i @ J_i]_{i=1}^n : \mathsf{Bag}\ A}\ \textsc{Result} \qquad \frac{[\vdash v_i : A_i]_{i=1}^n}{\vdash \langle \ell_i = v_i \rangle_{i=1}^n : \langle \ell_i = A_i \rangle_{i=1}^n}\ \textsc{Record} \qquad \frac{\Sigma(c) = O}{\vdash c : O}\ \textsc{Constant}$$

**Figure 16: Typing rules for nested annotated values**

$$\mathcal{S}\llbracket L \rrbracket = \mathcal{S}\llbracket L \rrbracket_{\varepsilon,1} \qquad \begin{array}{r} \mathcal{S}\llbracket \langle \ell = N \rangle_{i=1}^n \rrbracket_{\rho,\iota} = \langle \ell_i = \mathcal{S}\llbracket N_i \rrbracket_{\rho,\iota} \rangle_{i=1}^n \\ \mathcal{S}\llbracket X \rrbracket_{\rho,\iota} = \mathcal{N}\llbracket X \rrbracket_\rho \end{array} \qquad \begin{array}{r} \mathcal{S}\llbracket a \diamond \mathsf{out} \rrbracket_{\rho,\iota.i} = index(a \diamond \iota) \\ \mathcal{S}\llbracket a \diamond \mathsf{in} \rrbracket_{\rho,\iota.i} = index(a \diamond \iota.i) \end{array}$$

$$\mathcal{S}\llbracket \biguplus_{i=1}^n C_i \rrbracket_{\rho,\iota} = concat([\mathcal{S}\llbracket C_i \rrbracket_{\rho,\iota}]_{i=1}^n) \qquad \mathcal{S}\llbracket \mathsf{return}^a\ N \rrbracket_{\rho,\iota} = [\mathcal{S}\llbracket N \rrbracket_{\rho,\iota} @ index(a \diamond \iota)]$$

$$\mathcal{S}\llbracket \mathsf{for}\ ([x_i \leftarrow t_i]_{i=1}^n\ \mathsf{where}\ X)\ C \rrbracket_{\rho,\iota} = concat([\mathcal{S}\llbracket C \rrbracket_{\rho[x_i \mapsto r_i]_{i=1}^n, \iota.j} \mid \langle j, \vec{r} \rangle \leftarrow enum([\vec{r} \mid [r_i \leftarrow \llbracket t_i \rrbracket]_{i=1}^n, \mathcal{N}\llbracket X \rrbracket_{\rho[x_i \mapsto r_i]_{i=1}^n}])])$$

**Figure 17: Semantics of shredded queries, with annotations**

$$\begin{tikzcd}
& \llbracket \Sigma \rrbracket \arrow[dl, "\mathcal{N}\llbracket erase(L) \rrbracket"'] \arrow[d, "\mathcal{A}\llbracket L \rrbracket"] \arrow[dr, "\mathcal{H}\llbracket L \rrbracket_A"] & \\
\mathcal{N}\llbracket A \rrbracket & \mathcal{A}\llbracket A \rrbracket \arrow[l, "erase"] \arrow[r, bend left, "shred_{(-)}(A)"] & \mathcal{H}\llbracket A \rrbracket \arrow[l, bend left, "stitch"]
\end{tikzcd}$$

**Figure 15: Correctness of shredding and stitching**

terpretation of the input schema $\Sigma$ to the interpretation of its result type. To prove correctness is to prove that this diagram commutes.

## D.1 Annotated semantics of nested queries

We annotate bag elements with distinct indexes as follows:

| | |
|---|---|
| Results | $s ::= [w_1 @ I_1, \ldots, w_m @ I_m]$ |
| Inner values | $w ::= c \mid r \mid s$ |
| Rows | $r ::= \langle \ell_1 = w_1, \ldots, \ell_n = w_n \rangle$ |
| Indexes | $I, J$ |

Typing rules for these values are in Figure 16. The index annotations $@I$ on collection elements are needed solely for our correctness proof, and do not need to be present at run time.

The *canonical* choice for representing indexes is to take $I = a \diamond \iota$. We also allow for alternative indexing schemes in by parameterising the semantics over an indexing function $index$ mapping each canonical index $a \diamond \iota.j$ to a concrete representation. In the simplest case, we take $index$ to be the identity function (where static indexes are viewed as distinct integers). We will later consider other definitions of $index$ that depend on the particular query being shredded. Informally, the only constraints on $index$ are that it is defined on every canonical index $a \diamond \iota$ that we might need in order to shred a query, and it is injective on all canonical indexes. We will formalise these constraints later, but for now, $index$ can be assumed to be the identity function.

Given an indexing function $index$, the semantics of nested queries on annotated values is defined as follows:

$$\begin{array}{r}
\mathcal{A}\llbracket L \rrbracket = \mathcal{A}\llbracket L \rrbracket_{\varepsilon,1} \\
\mathcal{A}\llbracket \biguplus_{i=1}^n C_i \rrbracket_{\rho,\iota} = concat([\mathcal{A}\llbracket C_i \rrbracket_{\rho,\iota}]_{i=1}^n) \\
\mathcal{A}\llbracket \langle \ell_i = M_i \rangle_{i=1}^n \rrbracket_{\rho,\iota} = \langle \ell_i = \mathcal{A}\llbracket M_i \rrbracket_{\rho,\iota} \rangle_{i=1}^n \\
\mathcal{A}\llbracket X \rrbracket_{\rho,\iota} = \mathcal{N}\llbracket X \rrbracket_\rho
\end{array}$$

$$\begin{array}{l}
\mathcal{A}\llbracket \mathsf{for}\ ([x_i \leftarrow t_i]_{i=1}^n\ \mathsf{where}\ X)\ \mathsf{return}^a\ M \rrbracket_{\rho,\iota} = \\
\quad [\mathcal{A}\llbracket M \rrbracket_{\rho[x_i \mapsto r_i]_{i=1}^n, \iota.j} @ index(a \diamond \iota.j) \\
\qquad \mid \langle j, \vec{r} \rangle \leftarrow enum([\vec{r} \mid [r_i \leftarrow \llbracket t_i \rrbracket]_{i=1}^n, \mathcal{N}\llbracket X \rrbracket_{\rho[x_i \mapsto r_i]_{i=1}^n}])]
\end{array}$$

As well as an environment, the current dynamic index is threaded through the semantics. Note that we use the ordinary semantics to evaluate values that cannot have any annotations (such as boolean tests in the last case).

It is straightforward to show by induction that the annotated semantics erases to the standard semantics:

THEOREM 19. *For any $L$ we have $erase(\mathcal{A}\llbracket L \rrbracket) = \mathcal{N}\llbracket erase(L) \rrbracket$.*

## D.2 Shredding and stitching annotated values

To define the semantics of shredded queries and packages, we use annotated values in which collections are annotated pairs of indexes and annotated values. Again, we allow annotations $@J$ on elements of collections to facilitate the proof of correctness. Typing rules for these values are shown in Figure 18.

| | |
|---|---|
| Results | $s ::= [\langle I_1, w_1 \rangle @ J_1, \ldots, \langle I_m, w_m \rangle @ J_m]$ |
| Shredded values | $w ::= c \mid r \mid I$ |
| Rows | $r ::= \langle \ell_1 = w_1, \ldots, \ell_n = w_n \rangle$ |
| Indexes | $I, J$ |

Having defined suitably annotated versions of nested and shredded values, we now extend the shredding function to operate on nested values.

$$\begin{array}{r}
\llbracket s \rrbracket_p = \llbracket s \rrbracket^\star_{\top \diamond 1, p} \\
\llbracket [v_i @ J_i]_{i=1}^n \rrbracket^\star_{I,\epsilon} = [\langle I, \lfloor v_i \rfloor_{J_i} \rangle @ J_i]_{i=1}^n \\
\llbracket [v_i @ J_i]_{i=1}^n \rrbracket^\star_{I,\downarrow.p} = concat([\llbracket v_i \rrbracket^\star_{J_i,p}]_{i=1}^n) \\
\llbracket \langle \ell_i = v_i \rangle_{i=1}^n \rrbracket^\star_{I,\ell_i.p} = \llbracket v_i \rrbracket^\star_{I,p} \\
\lfloor c \rfloor_I = c \\
\lfloor \langle \ell_i = v_i \rangle_{i=1}^n \rfloor_I = \langle \ell_i = \lfloor v_i \rfloor_I \rangle_{i=1}^n \\
\lfloor s \rfloor_I = I
\end{array}$$

Note that in the cases for bags, the index annotation $J$ is passed as an argument to the recursive call: this is where we need the ghost indexes, to relate the semantics of nested and shredded queries.

We lift the nested result shredding function to build an (annotated) shredded value package in the same way that we did for

$$\frac{[\vdash I_i : \mathit{Index}]_{i=1}^n \qquad [\vdash w_i : F]_{i=1}^n \qquad [\vdash J_i : \mathit{Index}]_{i=1}^n}{\vdash \langle I_1, w_1 @ J_1\rangle, \dots, \langle I_n, w_n @ J_n\rangle : \mathrm{Bag}\, \langle \mathit{Index}, F\rangle}\ \textsc{Result} \qquad \frac{\Sigma(c) = O}{\vdash c : O}\ \textsc{Constant} \qquad \frac{[\vdash n_i : F_i]_{i=1}^n}{\vdash \langle \ell_i = w_i\rangle_{i=1}^n : \langle \ell_i = F_i\rangle_{i=1}^n}\ \textsc{Record}$$

**Figure 18: Typing rules for shredded values**

nested types and nested queries.

$$\begin{aligned} \mathit{shred}_s(\mathrm{Bag}\, A) &= \mathit{package}_{(\llbracket s \rrbracket_{-})}(\mathrm{Bag}\, A) \\ \mathit{shred}_{s,p}(\mathrm{Bag}\, A) &= \mathit{package}_{(\llbracket s \rrbracket_{-}),p}(\mathrm{Bag}\, A) \end{aligned}$$

We adjust the stitching function slightly to maintain annotations; the adjustment is consistent with its behaviour on unannotated values.

$$\begin{aligned} \mathit{stitch}(\hat{A}) &= \mathit{stitch}_{\top \diamond 1}(\hat{A}) \\ \mathit{stitch}_c(O) &= c \\ \mathit{stitch}_r(\langle \ell_i : \hat{A}_i\rangle_{i=1}^n) &= \langle \ell_i = \mathit{stitch}_{r.cl_i}(\hat{A}_i)\rangle_{i=1}^n \\ \mathit{stitch}_I((\mathrm{Bag}\, \hat{A})^s) &= [(\mathit{stitch}_w(\hat{A})) @ J \mid \langle I, w\rangle @ J \leftarrow s] \end{aligned}$$

The inner value parameter $w$ to the auxiliary function $\mathit{stitch}_w(-)$ specifies which values to stitch along the current path.

We now show how to interpret shredded queries and query packages over shredded values. The semantics of shredded queries (including annotations) is given in Figure 17. The semantics of a shredded query package is a *shredded value package* containing indexed results for each shredded query. For each type $A$ we define $\mathcal{H}\llbracket A \rrbracket = \mathit{shred}_A(A)$ and for each flat–nested, closed $\vdash L : A$ we define $\mathcal{H}\llbracket L \rrbracket_A : \mathcal{H}\llbracket A \rrbracket$ as $\mathit{pmap}_{\mathcal{S}\llbracket - \rrbracket}\ (\mathit{shred}_L(A))$.

## D.3 Main results

There are three key theorems. The first states that shredding commutes with the annotated semantics.

THEOREM 20. *If* $\vdash L : \mathrm{Bag}\, A$ *then:*

$$\mathcal{H}\llbracket L \rrbracket_{\mathrm{Bag}\, A} = \mathit{shred}_{\mathcal{A}\llbracket L \rrbracket}(\mathrm{Bag}\, A)$$

In order to allow shredded results to be correctly stitched together, we need the indexes at the end of each path to a bag through a nested value to be unique. We define $\mathit{indexes}_p(v)$, the indexes of nested value $v$ along path $p$ as follows.

$$\begin{aligned} \mathit{indexes}_\epsilon([v_i @ J_i]_{i=1}^n) &= [J_i]_{i=1}^n \\ \mathit{indexes}_{\downarrow.p}([v_i @ J_i]_{i=1}^n) &= \mathit{concat}([\mathit{indexes}_p(v_i)]_{i=1}^n) \\ \mathit{indexes}_{\ell_i.p}(\langle \ell_i = v_i\rangle_{i=1}^n) &= \mathit{indexes}_p(v_i) \end{aligned}$$

We say that a nested value $v$ is *well-indexed (at type $A$)* provided $\vdash v : A$, and for every path $p$ in $\mathit{paths}(A)$, the elements of $\mathit{indexes}_p(v)$ are distinct.

LEMMA 21. *If* $\vdash L : A$*, then* $\mathcal{A}\llbracket L \rrbracket$ *is well-indexed at* $A$*.*

Our next theorem states that for well-indexed values, stitching is a left-inverse of shredding.

THEOREM 22. *If* $\vdash s : \mathrm{Bag}\, A$ *and* $s$ *is well-indexed at type* $\mathrm{Bag}\, A$ *then* $\mathit{stitch}(\mathit{shred}_s(\mathrm{Bag}\, A)) = s$*.*

Combining Theorem 20, Lemma 21 and Theorem 22 we obtain the main correctness result (see Figure 15).

THEOREM 23. *If* $\vdash L : \mathrm{Bag}\, A$ *then:* $\mathit{stitch}(\mathcal{H}\llbracket L \rrbracket_{\mathrm{Bag}\, A}) = \mathcal{A}\llbracket L \rrbracket$*, and in particular:* $\mathit{erase}(\mathit{stitch}(\mathcal{H}\llbracket L \rrbracket_{\mathrm{Bag}\, A})) = \mathcal{N}\llbracket L \rrbracket$*.*

Not all possible indexing schemes satisfy Lemma 21. To identify those that do, we recall the function for computing the canonical indexes of a nested query.

$$\begin{aligned} \mathcal{I}\llbracket L \rrbracket &= \mathcal{I}\llbracket L \rrbracket_{\varepsilon,1} \\ \mathcal{I}\llbracket \biguplus_{i=1}^n C_i \rrbracket_{\rho,\iota} &= \mathit{concat}([\mathcal{I}\llbracket C_i \rrbracket_{\rho,\iota}]_{i=1}^n) \\ \mathcal{I}\llbracket \langle \ell_i = M_i\rangle_{i=1}^n \rrbracket_{\rho,\iota} &= \mathit{concat}([\mathcal{I}\llbracket M_i \rrbracket_{\rho,\iota}]_{i=1}^n) \\ \mathcal{I}\llbracket X \rrbracket_{\rho,\iota} &= [\,] \end{aligned}$$

$$\begin{aligned} &\mathcal{I}\llbracket \mathsf{for}\ ([x_i \leftarrow t_i]_{i=1}^n\ \mathsf{where}\ X)\ \mathsf{return}^a\ M \rrbracket_{\rho,\iota} = \\ &\quad \mathit{concat}([a \diamond \iota.j :: \mathcal{I}\llbracket M \rrbracket_{\rho[x_i \mapsto r_i]_{i=1}^n, \iota.j} \\ &\qquad \mid \langle j, \vec{r}\rangle \leftarrow \mathit{enum}([\vec{r} \mid [r_i \leftarrow \llbracket t_i \rrbracket]_{i=1}^n, \mathcal{N}\llbracket X \rrbracket_{\rho[x_i \mapsto r_i]_{i=1}^n}])]) \end{aligned}$$

Note that $\mathcal{I}\llbracket - \rrbracket$ resembles $\mathcal{A}\llbracket - \rrbracket$, but instead of the nested value $v$ it computes all indexes of $v$. An indexing function $\mathit{index} : \mathit{Index} \to A$ is *valid* with respect to the closed nested query $L$ if it is injective and defined on every canonical index in $\mathcal{I}\llbracket L \rrbracket$.

LEMMA 24. *If* $\mathit{index}$ *is valid for* $L$ *then* $\mathcal{A}\llbracket L \rrbracket$ *is well-indexed.*

The only requirement on indexes in the proof of Theorem 4 is that nested values be well-indexed, hence the proof extends to any valid indexing scheme. The concrete, natural, and flat indexing schemes are all valid.

## D.4 Detailed proofs

We first show that the inner shredding function $\llfloor - \rrfloor$ commutes with the annotated semantics.

LEMMA 25. $\mathcal{S}\llbracket \llfloor M \rrfloor_a \rrbracket_{\rho,\iota} = \llfloor \mathcal{A}\llbracket M \rrbracket_{\rho,\iota} \rrfloor_{a \diamond \iota}$

PROOF. By induction on the structure of $M$.

**Case** $x.\ell$**:**

$$\begin{aligned} & \mathcal{S}\llbracket \llfloor x.\ell \rrfloor_a \rrbracket_{\rho,\iota} \\ = \ & \mathcal{S}\llbracket x.\ell \rrbracket_{\rho,\iota} \\ = \ & \mathcal{A}\llbracket x.\ell \rrbracket_{\rho,\iota} \\ = \ & \llfloor \mathcal{A}\llbracket x.\ell \rrbracket_{\rho,\iota} \rrfloor_{a \diamond \iota} \end{aligned}$$

**Case** $c([X_i]_{i=1}^n)$**:**

$$\begin{aligned} & \mathcal{S}\llbracket \llfloor c([X_i]_{i=1}^n) \rrfloor_a \rrbracket_{\rho,\iota} \\ = \ & \mathcal{S}\llbracket c([\llfloor X_i \rrfloor_a]_{i=1}^n) \rrbracket_{\rho,\iota} \\ = \ & \mathcal{A}\llbracket c([\llfloor X_i \rrfloor_a]_{i=1}^n) \rrbracket_{\rho,\iota} \\ = \ & \llbracket c \rrbracket([\mathcal{A}\llbracket \llfloor X_i \rrfloor_a \rrbracket_{\rho,\iota}]_{i=1}^n) \\ = \ & \llfloor \mathcal{A}\llbracket c([X_i]_{i=1}^n) \rrbracket_{\rho,\iota} \rrfloor_{a \diamond \iota} \end{aligned}$$

**Case** empty $M$**:**

$$\mathcal{S}[\![\lfloor\mathsf{empty}\ M\rfloor_a]\!]_{\rho,\iota}$$
$=$
$$\mathcal{S}[\![\mathsf{empty}\ \lceil M\rceil_\epsilon]\!]_{\rho,\iota}$$
$=$
$$\mathcal{A}[\![\mathsf{empty}\ \lceil M\rceil_\epsilon]\!]_{\rho,\iota}$$
$=$
$$\lfloor\mathcal{A}[\![\mathsf{empty}\ M]\!]_{\rho,\iota}\rfloor_{a\diamond\iota}$$

**Case $\langle\overrightarrow{\ell = M}\rangle$:**

$$\mathcal{S}[\![\lfloor\langle\ell_i = M_i\rangle_{i=1}^n\rfloor_a]\!]_{\rho,\iota}$$
$=$
$$\mathcal{S}[\![\langle\ell_i = \lfloor M_i\rfloor_a\rangle_{i=1}^n]\!]_{\rho,\iota}$$
$=$
$$\langle\ell_i = \mathcal{S}[\![\lfloor M_i\rfloor_a]\!]_{\rho,\iota}\rangle_{i=1}^n$$
$=$ (Induction hypothesis)
$$\langle\ell_i = \lfloor\mathcal{A}[\![M_i]\!]_{\rho,\iota}\rfloor_{a\diamond\iota}\rangle_{i=1}^n$$
$=$
$$\lfloor\langle\ell_i = \mathcal{A}[\![M_i]\!]_{\rho,\iota}\rangle_{i=1}^n\rfloor_{a\diamond\iota}$$
$=$
$$\lfloor\mathcal{A}[\![\langle\ell_i = M_i\rangle_{i=1}^n]\!]_{\rho,\iota}\rfloor_{a\diamond\iota}$$

**Case $L$:**

$$\mathcal{S}[\![\lfloor L\rfloor_a]\!]_{\rho,\iota}$$
$=$
$$\mathcal{S}[\![a\diamond\mathsf{in}]\!]_{\rho,\iota}$$
$=$
$$a\diamond\iota$$
$=$
$$\lfloor\mathcal{A}[\![L]\!]_{\rho,\iota}\rfloor_{a\diamond\iota}$$

This completes the proof. $\square$

The first part of the following lemma allows us to run a shredded query by concatenating the results of running the shreddings of its component comprehensions. Similarly, the second part allows us to shred the results of running a nested query by concatenating the shreddings of the results of running its component comprehensions.

LEMMA 26.
*1.* $\mathcal{S}[\![\biguplus\lceil\biguplus_{i=1}^n C_i\rceil_{a,p}^\star]\!]_{\rho,\iota} = concat([\mathcal{S}[\![\lceil C_i\rceil_{a,p}^\star]\!]_{\rho,\iota}]_{i=1}^n)$
*2.* $\lceil\mathcal{A}[\![\biguplus_{i=1}^n C_i]\!]_{\rho,\iota}\rceil_{a\diamond\iota,p}^\star = concat([\lceil\mathcal{A}[\![C_i]\!]_{\rho,\iota}\rceil_{a\diamond\iota,p}^\star]_{i=1}^n)$

PROOF.
1.

$$\mathcal{S}[\![\biguplus\lceil\biguplus_{i=1}^n C_i\rceil_{a,p}^\star]\!]_{\rho,\iota}$$
$=$
$$\mathcal{S}[\![\biguplus\ concat([\lceil C_i\rceil_{a,p}^\star]_{i=1}^n)]\!]_{\rho,\iota}$$
$=$
$$\mathcal{S}[\![\biguplus\ concat([\lceil C\rceil_{a,p}^\star \mid C \leftarrow \vec{C}])]\!]_{\rho,\iota}$$
$=$
$$concat(concat([\mathcal{S}[\![C']\!]_{\rho,\iota} \mid C \leftarrow \vec{C}, C' \leftarrow \lceil C\rceil_{a,p}^\star]))$$
$=$
$$concat([\mathcal{S}[\![\biguplus\lceil C\rceil_{a,p}^\star]\!]_{\rho,\iota} \mid C \leftarrow \vec{C}])$$
$=$
$$concat([\mathcal{S}[\![\biguplus\lceil C_i\rceil_{a,p}^\star]\!]_{\rho,\iota}]_{i=1}^n)$$

2.

$$\lceil\mathcal{A}[\![\biguplus_{i=1}^n C_i]\!]_{\rho,\iota}\rceil_{a\diamond\iota,p}^\star$$
$=$
$$\lceil concat([\mathcal{A}[\![C_i]\!]_{\rho,\iota}]_{i=1}^n)\rceil_{a\diamond\iota,p}^\star$$
$=$
$$\lceil concat([\mathcal{A}[\![C]\!]_{\rho,\iota} \mid C \leftarrow \vec{C}])\rceil_{a\diamond\iota,p}^\star$$
$=$
$$concat(concat([\lceil s\rceil_{a\diamond\iota,p}^\star \mid C \leftarrow \vec{C}, s \leftarrow \mathcal{A}[\![C]\!]_{\rho,\iota}]))$$
$=$
$$concat([\lceil\mathcal{A}[\![C]\!]_{\rho,\iota}\rceil_{a\diamond\iota,p}^\star \mid C \leftarrow \vec{C}])$$
$=$
$$concat([\lceil\mathcal{A}[\![C_i]\!]_{\rho,\iota}\rceil_{a\diamond\iota,p}^\star]_{i=1}^n)$$

This completes the proof. $\square$

We are now in a position to prove that the outer shredding function $\lceil - \rceil$ commutes with the semantics.

LEMMA 27. $\mathcal{S}[\![\lceil L\rceil_p]\!]_{\rho,1} = \lceil\mathcal{A}[\![L]\!]_{\rho,1}\rceil_p$

PROOF. We prove the following:
1. $\mathcal{S}[\![\lceil L\rceil_p]\!]_{\rho,1} = \lceil\mathcal{A}[\![L]\!]_{\rho,1}\rceil_p$
2. $\mathcal{S}[\![\biguplus\lceil C\rceil_{a,p}^\star]\!]_{\rho,\iota} = \lceil\mathcal{A}[\![C]\!]_{\rho,\iota}\rceil_{a\diamond\iota,p}^\star$
3. $\mathcal{S}[\![\lceil M\rceil_{a,p}^\star]\!]_{\rho,\iota} = \lceil\mathcal{A}[\![M]\!]_{\rho,\iota}\rceil_{a\diamond\iota,p}^\star$

The first equation is the result we require. Observe that it follows from (2) and Lemma 26. We now proceed to prove equations (2) and (3) by mutual induction on the structure of $p$.

There are only two cases for (2), as the $\ell_i.p$ case cannot apply.

**Case $\epsilon$:**

$$\mathcal{S}[\![\biguplus\lceil\mathsf{for}\ (\overrightarrow{x \leftarrow t}\ \mathsf{where}\ X)\ \mathsf{return}^b\ M\rceil_{a,\epsilon}^\star]\!]_{\rho,\iota}$$
$=$ (Definition of $\mathcal{S}[\![-]\!]$)
$$\mathcal{S}[\![\mathsf{for}\ (\overrightarrow{x \leftarrow t}\ \mathsf{where}\ X)\ \mathsf{return}^b\ \langle a\diamond\iota, \lfloor M\rfloor_b\rangle]\!]_{\rho,\iota}$$
$=$ (Definition of $\mathcal{S}[\![-]\!]$)
$$[\langle a\diamond\iota, \mathcal{S}[\![\lfloor M\rfloor_b]\!]_{\rho[\overrightarrow{x\mapsto v}],\iota.i}\rangle @ b\diamond\iota.i \mid \langle i,\vec{v}\rangle \leftarrow enum([\vec{v} \mid \overrightarrow{v \leftarrow [\![t]\!]}, \mathcal{S}[\![X]\!]_{\rho[\overrightarrow{x\mapsto v}]}])]$$
$=$ (Lemma 25)
$$[\langle a\diamond\iota, \lfloor\mathcal{A}[\![M]\!]_{\rho[\overrightarrow{x\mapsto v}],\iota.i}\rfloor_{b\diamond\iota.i}\rangle @ b\diamond\iota.i \mid \langle i,\vec{v}\rangle \leftarrow enum([\vec{v} \mid \overrightarrow{v \leftarrow [\![t]\!]}, \mathcal{S}[\![X]\!]_{\rho[\overrightarrow{x\mapsto v}]}])]$$
$=$ (Definition of $\lceil-\rceil^\star$)
$$\lceil[\mathcal{A}[\![M]\!]_{\rho[\overrightarrow{x\mapsto v}],\iota.i} @ b\diamond\iota.i \mid \langle i,\vec{v}\rangle \leftarrow enum([\vec{v} \mid \overrightarrow{v \leftarrow [\![t]\!]}, \mathcal{S}[\![X]\!]_{\rho[\overrightarrow{x\mapsto v}]}])]\rceil_{a\diamond\iota,\epsilon}^\star$$
$=$ (Definition of $\mathcal{A}[\![-]\!]$)
$$\lceil\mathcal{A}[\![\mathsf{for}\ (\overrightarrow{x \leftarrow t}\ \mathsf{where}\ X)\ \mathsf{return}^b\ M]\!]_{\rho,\iota}\rceil_{a\diamond\iota,\epsilon}^\star$$

**Case $\downarrow.\epsilon$:**

$$
\begin{array}{ll}
 & \mathcal{S}\llbracket \biguplus \lceil\!\lceil \mathsf{for}\,(\overrightarrow{x \leftarrow t}\,\mathsf{where}\,X)\,\mathsf{return}^b\, M \rceil\!\rceil^\star_{a,\downarrow.p} \rrbracket_{\rho,\iota} \\
= & \quad \text{(Definition of } \lceil\!\lceil - \rceil\!\rceil^\star) \\
 & \mathcal{S}\llbracket \biguplus [\mathsf{for}\,(\overrightarrow{x \leftarrow t}\,\mathsf{where}\,X)\, C \mid C \leftarrow \lceil\!\lceil M \rceil\!\rceil^\star_{b,p}] \rrbracket_{\rho,\iota} \\
= & \quad \text{(Definition of } \mathcal{S}\llbracket - \rrbracket) \\
 & \mathit{concat}([\mathit{concat} \\
 & \qquad ([\mathcal{S}\llbracket C \rrbracket_{\rho[\overrightarrow{x \mapsto v}],\iota.i} \\
 & \qquad\quad \mid \langle i, \vec{v} \rangle \leftarrow \mathit{enum}([\vec{v} \mid \overrightarrow{v \leftarrow \llbracket t \rrbracket}, \mathcal{S}\llbracket X \rrbracket_{\rho[\overrightarrow{x \mapsto v}]}])]) \\
 & \qquad \mid C \leftarrow \lceil\!\lceil M \rceil\!\rceil^\star_{b,p}]) \\
= & \quad \text{(Definition of } \mathcal{S}\llbracket - \rrbracket) \\
 & \mathit{concat} \\
 & \quad ([\mathcal{S}\llbracket \lceil\!\lceil M \rceil\!\rceil^\star_{b,p} \rrbracket_{\rho[\overrightarrow{x \mapsto v}],\iota.i} \\
 & \qquad \mid \langle i, \vec{v} \rangle \leftarrow \mathit{enum}([\vec{v} \mid \overrightarrow{v \leftarrow \llbracket t \rrbracket}, \mathcal{S}\llbracket X \rrbracket_{\rho[\overrightarrow{x \mapsto v}]}])]) \\
= & \quad \text{(Induction hypothesis (3))} \\
 & \mathit{concat} \\
 & \quad ([\lceil\!\lceil \mathcal{A}\llbracket M \rrbracket_{\rho[\overrightarrow{x \mapsto v}],\iota.i} \rceil\!\rceil^\star_{b,p} \\
 & \qquad \mid \langle i, \vec{v} \rangle \leftarrow \mathit{enum}([\vec{v} \mid \overrightarrow{v \leftarrow \llbracket t \rrbracket}, \mathcal{S}\llbracket X \rrbracket_{\rho[\overrightarrow{x \mapsto v}]}])]) \\
= & \quad \text{(Definitions of } \mathcal{A}\llbracket - \rrbracket \text{ and } \lceil\!\lceil - \rceil\!\rceil^\star) \\
 & \lceil\!\lceil \mathcal{A}\llbracket \mathsf{for}\,(\overrightarrow{x \leftarrow t}\,\mathsf{where}\,X)\,\mathsf{return}^b\, M \rrbracket_{\rho,\iota} \rceil\!\rceil^\star_{a \diamond \iota, \downarrow.p}
\end{array}
$$

There are three cases for (3).

**Cases $\epsilon$ and $\downarrow.\epsilon$:** follow from (2) by applying the two parts of Lemma 26 to the left and right-hand side respectively.

**Case $\ell_i.\epsilon$:**

$$
\begin{array}{ll}
 & \mathcal{S}\llbracket \lceil\!\lceil \langle \overrightarrow{\ell = M} \rangle \rceil\!\rceil^\star_{a,\ell_i.p} \rrbracket_{\rho,\iota} \\
= & \quad \text{(Definition of } \lceil\!\lceil - \rceil\!\rceil^\star) \\
 & \mathcal{S}\llbracket \lceil\!\lceil M_i \rceil\!\rceil^\star_{a,p} \rrbracket_{\rho,\iota} \\
= & \quad \text{(Induction hypothesis (3))} \\
 & \lceil\!\lceil \mathcal{A}\llbracket M_i \rrbracket_{\rho,\iota} \rceil\!\rceil^\star_{a \diamond \iota, p} \\
= & \quad \text{(Definition of } \mathcal{A}\llbracket - \rrbracket) \\
 & \lceil\!\lceil \mathcal{A}\llbracket \langle \overrightarrow{\ell = M} \rangle \rrbracket_{\rho,\iota} \rceil\!\rceil^\star_{a \diamond \iota, \ell_i.p}
\end{array}
$$

This completes the proof. $\square$

We now lift Lemma 27 to shredded packages.

PROOF OF THEOREM 20. We need to show that if $\vdash L : \mathrm{Bag}\ A$ then

$$\mathcal{H}\llbracket L \rrbracket_A = \mathit{shred}_{\mathcal{A}\llbracket L \rrbracket}(\mathrm{Bag}\ A)$$

This is straightforward by induction on $A$, using Lemma 27. $\square$

We have proved that shredding commutes with the semantics. It remains to show that stitching after shredding is the identity on index-annotated nested results. We need two auxiliary notions: the *descendant* of a value at a path, and the *indexes* at the end of a path (which must be unique in order for stitching to work).

We define $/\!\!/ v \backslash\!\!\backslash_{p,w}$, the *descendant* of a value $v$ at path $p$ with respect to inner value $w$ as follows.

$$
\begin{array}{rcl}
/\!\!/ v \backslash\!\!\backslash_{p,w} & = & /\!\!/ v \backslash\!\!\backslash_{\top \diamond 1, p, w} \\
/\!\!/ v \backslash\!\!\backslash_{J,p,c} & = & c \\
/\!\!/ v \backslash\!\!\backslash_{J,p,\langle \ell_i = w_i \rangle_{i=1}^n} & = & \langle \ell_i = /\!\!/ v \backslash\!\!\backslash_{J,p.\ell_i,w_i} \rangle_{i=1}^n \\
/\!\!/ s \backslash\!\!\backslash_{J,\epsilon,I} & = & \left\{ \begin{array}{ll} s, & \text{if } J = I \\ [\,], & \text{if } J \neq I \end{array} \right. \\
/\!\!/ [v_i @ J_i]_{i=1}^n \backslash\!\!\backslash_{J,\downarrow.p,I} & = & \mathit{concat}([/\!\!/ v_i \backslash\!\!\backslash_{J_i,p,I}]_{i=1}^n) \\
/\!\!/ \langle \ell_i = v_i \rangle_{i=1}^n \backslash\!\!\backslash_{J,\ell_i.p,I} & = & /\!\!/ v_i \backslash\!\!\backslash_{J,p,I}
\end{array}
$$

Essentially, this extracts the part of $v$ that corresponds to $w$. The inner value $w$ allows us to specify a particular descendant as an index, or nested record of indexes; for uniformity it may also contain constants.

We can now formulate the following crucial technical lemma, which states that given the descendants of a result $v$ at path $p.\downarrow$ and the shredded values of $v$ at path $p$ we can stitch them together to form the descendants at path $p$.

LEMMA 28 (KEY LEMMA). *If $v$ is well-indexed and $\vdash /\!\!/ v \backslash\!\!\backslash_{J,p,I}$ : $\mathrm{Bag}\ A$, then*

$$/\!\!/ v \backslash\!\!\backslash_{J,p,I} = [/\!\!/ v \backslash\!\!\backslash_{J,p.\downarrow,w} @ I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, w \rangle @ I_{\mathsf{in}} \leftarrow \lceil\!\lceil v \rceil\!\rceil^\star_{J,p}, I_{\mathsf{out}} = I]$$

PROOF. First we strengthen the induction hypothesis to account for records. The generalised induction hypothesis is as follows.

If $v$ is well-indexed and $\vdash /\!\!/ v \backslash\!\!\backslash_{J',p.\vec{\ell},I}$ : $\mathrm{Bag}\ A$, then

$$
\begin{array}{l}
[/\!\!/ v \backslash\!\!\backslash_{J',p.\downarrow.\vec{\ell},w} @ I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, w \rangle @ I_{\mathsf{in}} \leftarrow \lceil\!\lceil v \rceil\!\rceil^\star_{J',p.\vec{\ell}}, I_{\mathsf{out}} = I] \\
\qquad = [v'.\vec{\ell} @ I_{\mathsf{in}} \mid v' @ I_{\mathsf{in}} \leftarrow /\!\!/ v \backslash\!\!\backslash_{J',p,I}]
\end{array}
$$

The proof now proceeds by induction on the structure of $A$ and side-induction on the structure of $p$.

**Case $O$:**

**Subcase $\epsilon$:** If $J' \neq I$ then both sides are empty lists. Suppose that $J' = I$.

$$
\begin{array}{ll}
 & [(/\!\!/ s \backslash\!\!\backslash_{J',\downarrow.\vec{\ell},c} @ I_{\mathsf{in}} \\
 & \qquad \mid \langle I_{\mathsf{out}}, c \rangle @ I_{\mathsf{in}} \leftarrow \lceil\!\lceil s \rceil\!\rceil^\star_{J',\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad \text{(Definition of } /\!\!/ - \backslash\!\!\backslash) \\
 & [c @ I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, c \rangle @ I_{\mathsf{in}} \leftarrow \lceil\!\lceil s \rceil\!\rceil^\star_{J',\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad (J' = I) \\
 & s \\
= & \quad \text{(Definition of } /\!\!/ - \backslash\!\!\backslash) \\
 & [v'.\vec{\ell} @ I_{\mathsf{in}} \mid v' @ I_{\mathsf{in}} \leftarrow /\!\!/ s \backslash\!\!\backslash_{J',\epsilon,I}]
\end{array}
$$

**Subcase $\ell_i.p$:**

$$
\begin{array}{ll}
 & [(/\!\!/ \langle \overrightarrow{\ell = v} \rangle \backslash\!\!\backslash_{J',\ell_i.p.\downarrow.\vec{\ell},c} @ I_{\mathsf{in}} \\
 & \qquad \mid \langle I_{\mathsf{out}}, c @ I_{\mathsf{in}} \rangle \leftarrow \lceil\!\lceil \langle \overrightarrow{\ell = v} \rangle \rceil\!\rceil^\star_{J',\ell_i.p.\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad \text{(Definition of } /\!\!/ - \backslash\!\!\backslash) \\
 & [c @ I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, c @ I_{\mathsf{in}} \rangle \leftarrow \lceil\!\lceil \langle \overrightarrow{\ell = v} \rangle \rceil\!\rceil^\star_{J',\ell_i.p.\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad \text{(Definition of } \lceil\!\lceil - \rceil\!\rceil^\star) \\
 & [c @ I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, c @ I_{\mathsf{in}} \rangle \leftarrow \lceil\!\lceil v_i \rceil\!\rceil^\star_{J',p.\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad \text{(Definition of } /\!\!/ - \backslash\!\!\backslash) \\
 & [(/\!\!/ v_i \backslash\!\!\backslash_{J',p.\downarrow.\vec{\ell},c} @ I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, c \rangle @ I_{\mathsf{in}} \leftarrow \lceil\!\lceil v_i \rceil\!\rceil^\star_{J',p.\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad \text{(Induction hypothesis)} \\
 & [v'.\vec{\ell} @ I_{\mathsf{in}} \mid v' @ I_{\mathsf{in}} \leftarrow /\!\!/ v_i \backslash\!\!\backslash_{J',p,I}] \\
= & \quad \text{(Definition of } \lceil\!\lceil - \rceil\!\rceil^\star) \\
 & [v'.\vec{\ell} @ I_{\mathsf{in}} \mid v' @ I_{\mathsf{in}} \leftarrow /\!\!/ \langle \overrightarrow{\ell = v} \rangle \backslash\!\!\backslash_{J',\ell_i.p,I}]
\end{array}
$$

**Subcase $\downarrow.p$:**

$$
\begin{array}{ll}
 & [(/\!\!/ [v_i @ J_i]_{i=1}^n \backslash\!\!\backslash_{J',\downarrow.p.\downarrow.\vec{\ell},c}) @ I_{\mathsf{in}} \\
 & \qquad \mid \langle I_{\mathsf{out}}, c \rangle @ I_{\mathsf{in}} \leftarrow \lceil\!\lceil [v_i @ J_i]_{i=1}^n \rceil\!\rceil^\star_{J',\downarrow.p.\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad \text{(Definitions of } /\!\!/ - \backslash\!\!\backslash \text{ and } \lceil\!\lceil - \rceil\!\rceil^\star) \\
 & \mathit{concat}([c @ I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, c @ I_{\mathsf{in}} \rangle \leftarrow \lceil\!\lceil v_i \rceil\!\rceil^\star_{J_i,p.\vec{\ell}}, I_{\mathsf{out}} = I]_{i=1}^n) \\
= & \quad \text{(Induction hypothesis)} \\
 & \mathit{concat}([v'.\vec{\ell} @ I_{\mathsf{in}} \mid v' @ I_{\mathsf{in}} \leftarrow /\!\!/ v_i \backslash\!\!\backslash_{J_i,p}]_{i=1}^n) \\
= & \quad \text{(Definition of } /\!\!/ - \backslash\!\!\backslash) \\
 & [v'.\vec{\ell} @ I_{\mathsf{in}} \mid v' @ I_{\mathsf{in}} \leftarrow /\!\!/ [v @ J]_{i=1}^n \backslash\!\!\backslash_{J',p}]
\end{array}
$$

**Case $\langle\rangle$:** The proof is the same as for base types with the constant $c$ replaced by $\langle\rangle$.

**Case $\langle \overrightarrow{\ell : A}\rangle$ where $|\vec{\ell}| \geq q$:** We rely on the functions $zip_{\vec{\ell}}$, for transforming a record of lists of equal length to a list of records, and $unzip_{\vec{\ell}}$, for transforming a list of records to a record of lists of equal length. In fact we require special versions of $zip$ and $unzip$ that handle annotations, such that $zip$ takes a record of lists of equal length whose annotations must be in sync, and $unzip$ returns such a record.

$$\begin{array}{rcl}
zip_{\vec{\ell}}\,\langle \ell_i = []\rangle_{i=1}^n &=& [] \\
zip_{\vec{\ell}}\,\langle \ell_i = v_i@J :: s_i\rangle_{i=1}^n &=& \langle \ell_i = v_i\rangle_{i=1}^n@J :: zip_{\vec{\ell}}\,\langle \ell_i = s_i\rangle_{i=1}^n \\
unzip_{\vec{\ell}}(s) &=& \langle \ell_i = [v.\ell_i@J \mid v@J \leftarrow s]\rangle_{i=1}^n
\end{array}$$

If $\vec{\ell}$ is a non-empty list of column labels then $zip_{\vec{\ell}}$ is the inverse of $unzip_{\vec{\ell}}$.

$$zip_{\vec{\ell}}(unzip_{\vec{\ell}}(s)) = s, \qquad \text{if } |\vec{\ell}| \geq 1$$

$$\begin{array}{ll}
& [/\!\!/ v \backslash\!\!\backslash_{J',p.\downarrow.\vec{\ell'},w}@I_{\mathsf{in}} \\
& \quad \mid \langle I_{\mathsf{out}}, w\rangle@I_{\mathsf{in}} \leftarrow [\![v]\!]^{\star}_{J',p.\vec{\ell'}}, I_{\mathsf{out}} = I] \\
= & \quad (\text{Definition of } /\!\!/-\backslash\!\!\backslash) \\
& [\langle \ell_i = /\!\!/ v \backslash\!\!\backslash_{J',p.\downarrow.\vec{\ell'}.\ell_i,w}\rangle_{i=1}^n@I_{\mathsf{in}} \\
& \quad \mid \langle I_{\mathsf{out}}, w@I_{\mathsf{in}}\rangle \leftarrow [\![v]\!]^{\star}_{J',p.\vec{\ell'}.\ell}, I_{\mathsf{out}} = I] \\
= & \quad (\text{Definition of } zip) \\
& zip_{\vec{\ell}}\,\langle \ell_i = [v_{\ell_i,w}@I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, w@I_{\mathsf{in}}\rangle \leftarrow [\![v]\!]^{\star}_{J',p.\vec{\ell'}.\ell_i}, \\
& \qquad\qquad\qquad\qquad\qquad\qquad I_{\mathsf{out}} = I]\rangle_{i=1}^n \\
& \qquad \text{where } v_{\ell,w} \text{ stands for } /\!\!/ v \backslash\!\!\backslash_{J',p.\downarrow.\vec{\ell'}.\ell,w} \\
= & \quad (\text{Induction hypothesis}) \\
& zip_{\vec{\ell}}\,\langle \ell_i = [v'.\vec{\ell'}.\ell_i@I_{\mathsf{in}} \mid v'@I_{\mathsf{in}} \leftarrow /\!\!/ v \backslash\!\!\backslash_{J',p,I}]\rangle_{i=1}^n \\
= & \quad (\text{Definition of } zip) \\
& [v'.\vec{\ell'}@I_{\mathsf{in}} \mid v'@I_{\mathsf{in}} \leftarrow /\!\!/ v \backslash\!\!\backslash_{J',p,I}]
\end{array}$$

**Case $\mathrm{Bag}\,\vec{A}$: Subcase $\epsilon$:** If $J' \neq I$ then both sides of the equation are equivalent to the empty bag. Suppose $J' = I$.

$$\begin{array}{ll}
& [/\!\!/ [v_i@J_i]_{i=1}^n \backslash\!\!\backslash_{J',\downarrow.\vec{\ell},I_{\mathsf{in}}}@I_{\mathsf{in}} \\
& \quad \mid \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow [\![[v_i@J_i]_{i=1}^n]\!]^{\star}_{J',\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad (\text{Definition of } /\!\!/-\backslash\!\!\backslash) \\
& [concat([/\!\!/ v_i \backslash\!\!\backslash_{J_i,\vec{\ell},I_{\mathsf{in}}}]_{i=1}^n)@I_{\mathsf{in}} \\
& \quad \mid \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow [\![[v_i@J_i]_{i=1}^n]\!]^{\star}_{J',\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad ([\![[v_i@J_i]_{i=1}^n]\!]^{\star}_{J',\vec{\ell}} = \langle J', J_1\rangle@J_1, \ldots, \langle J', J_n\rangle@J_n) \\
& [concat([/\!\!/ v_i \backslash\!\!\backslash_{J_i,\vec{\ell},I_{\mathsf{in}}}]_{i=1}^n)@I_{\mathsf{in}} \mid I_{\mathsf{in}} \leftarrow \vec{J}, J' = I] \\
= & \quad (J' = I) \\
& [concat([/\!\!/ v_i \backslash\!\!\backslash_{J_i,\vec{\ell},I_{\mathsf{in}}}]_{i=1}^n)@I_{\mathsf{in}} \mid I_{\mathsf{in}} \leftarrow \vec{J}] \\
= & \quad (\text{Definition of } /\!\!/-\backslash\!\!\backslash) \\
& [concat([/\!\!/ v_i \backslash\!\!\backslash_{J_i,\vec{\ell},I_{\mathsf{in}}} \\
& \qquad\qquad \mid v_i@J_i \leftarrow [v_i@J_i]_{i=1}^n])@I_{\mathsf{in}} \mid I_{\mathsf{in}} \leftarrow \vec{J}] \\
= & \quad (\text{Definition of } /\!\!/-\backslash\!\!\backslash) \\
& [concat([/\!\!/ v_i.\vec{\ell} \backslash\!\!\backslash_{J_i,\epsilon,I_{\mathsf{in}}} \mid v_i@J_i \leftarrow [v_i@J_i]_{i=1}^n])@I_{\mathsf{in}} \\
& \qquad\qquad\qquad\qquad \mid I_{\mathsf{in}} \leftarrow \vec{J}] \\
= & \quad (\text{Definition of } /\!\!/-\backslash\!\!\backslash) \\
& [concat([v_i.\vec{\ell} \\
& \qquad \mid v_i@J_i \leftarrow [v_i@J_i]_{i=1}^n, J_i = I_{\mathsf{in}}])@I_{\mathsf{in}} \mid I_{\mathsf{in}} \leftarrow \vec{J}] \\
= & \quad (v \text{ is well-indexed}) \\
& [v_i.\vec{\ell}@I_{\mathsf{in}} \mid v_i@I_{\mathsf{in}} \leftarrow [v_i@J_i]_{i=1}^n] \\
= & \quad (\text{Definition of } /\!\!/-\backslash\!\!\backslash \text{ and } J' = I) \\
& [v_i.\vec{\ell}@I_{\mathsf{in}} \mid v_i@I_{\mathsf{in}} \leftarrow /\!\!/ [v_i@J_i]_{i=1}^n \backslash\!\!\backslash_{J',\epsilon,I}]
\end{array}$$

**Subcase $\ell_i.p$:**

$$\begin{array}{ll}
& [/\!\!/ \langle \ell_i = v_i\rangle_{i=1}^n \backslash\!\!\backslash_{J',\ell_i.p.\downarrow.\vec{\ell'},I_{\mathsf{in}}}@I_{\mathsf{in}} \\
& \quad \mid \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow [\![\langle \ell_i = v_i\rangle_{i=1}^n]\!]^{\star}_{J',\ell_i.p.\vec{\ell'}}, I_{\mathsf{out}} = I] \\
= & \quad (\text{Definitions of } /\!\!/-\backslash\!\!\backslash \text{ and } [\![-]\!]^{\star}) \\
& [/\!\!/ v_i \backslash\!\!\backslash_{J',p.\downarrow.\vec{\ell'},I_{\mathsf{in}}}@I_{\mathsf{in}} \\
& \quad \mid \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow [\![v_i]\!]^{\star}_{J',p.\vec{\ell'}}, I_{\mathsf{out}} = I] \\
= & \quad (\text{Induction hypothesis}) \\
& [v'.\vec{\ell'}@I_{\mathsf{in}} \mid v'@I_{\mathsf{in}} \leftarrow /\!\!/ v_i \backslash\!\!\backslash_{J',p,I}] \\
= & \quad (\text{Definition of } /\!\!/-\backslash\!\!\backslash) \\
& [v'.\vec{\ell'}@I_{\mathsf{in}} \mid v'@I_{\mathsf{in}} \leftarrow /\!\!/ \langle \ell_i = v_i\rangle_{i=1}^n \backslash\!\!\backslash_{J',\ell_i.p,I}]
\end{array}$$

**Subcase $\downarrow.p$:**

$$\begin{array}{ll}
& [/\!\!/ [v_i@J_i]_{i=1}^n \backslash\!\!\backslash_{J',\downarrow.p.\downarrow.\vec{\ell},I_{\mathsf{in}}}@I_{\mathsf{in}} \\
& \quad \mid \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow [\![[v_i@J_i]_{i=1}^n]\!]^{\star}_{I_{\mathsf{out}},\downarrow.p.\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad (\text{Definitions of } /\!\!/-\backslash\!\!\backslash \text{ and } [\![-]\!]^{\star}) \\
& [(concat([/\!\!/ v_i \backslash\!\!\backslash_{J_i,p.\downarrow.\vec{\ell},I_{\mathsf{in}}}]))@I_{\mathsf{in}} \\
& \quad \mid \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow concat([[\![v_i]\!]^{\star}_{J_i,p.\vec{\ell}}]_{i=1}^n), I_{\mathsf{out}} = I] \\
= & \quad (\text{Expanding } concat([[\![-]\!]^{\star}]_{i=1}^n)) \\
& [(concat([/\!\!/ v_i \backslash\!\!\backslash_{J_i,p.\downarrow.\vec{\ell},I_{\mathsf{in}}}]))@I_{\mathsf{in}} \\
& \quad \mid v_i@J_i \leftarrow [v_i@J_i]_{i=1}^n, \\
& \qquad \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow [\![v_i]\!]^{\star}_{J_i,p.\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad (v \text{ is well-indexed}) \\
& [/\!\!/ v_i \backslash\!\!\backslash_{J_i,p.\downarrow.\vec{\ell},I_{\mathsf{in}}}@I_{\mathsf{in}} \\
& \quad \mid v_i@J_i \leftarrow [v_i@J_i]_{i=1}^n, \\
& \qquad \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow [\![v_i]\!]^{\star}_{J_i,p.\vec{\ell}}, I_{\mathsf{out}} = I] \\
= & \quad (\text{Comprehension} \rightarrow \text{concatenation}) \\
& concat \\
& \quad ([/\!\!/ v_i \backslash\!\!\backslash_{J_i,p.\downarrow.\vec{\ell},I_{\mathsf{in}}}@I_{\mathsf{in}} \\
& \qquad \mid \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow [\![v_i]\!]^{\star}_{J_i,p.\vec{\ell}}, I_{\mathsf{out}} = I]_{i=1}^n) \\
= & \quad (\text{Induction hypothesis}) \\
& concat([v'.\vec{\ell}@I_{\mathsf{in}} \mid v'@I_{\mathsf{in}} \leftarrow /\!\!/ v_i \backslash\!\!\backslash_{J_i,\downarrow.p,I}]_{i=1}^n) \\
= & \quad (\text{Concatenation} \rightarrow \text{comprehension}) \\
& [v'.\vec{\ell}@I_{\mathsf{in}} \mid v'@I_{\mathsf{in}} \leftarrow /\!\!/ [v_i@J_i]_{i=1}^n \backslash\!\!\backslash_{J',\downarrow.p,I}]
\end{array}$$

This completes the proof. □

The proof of this lemma is the only part of the formalisation that makes use of values being well-indexed. Stitching shredded results together does not depend on any other property of indexes, thus any representation of indexes that yields unique indexes suffices.

THEOREM 29. *If $s$ is well-indexed and $\vdash /\!\!/ s \backslash\!\!\backslash_{p,w} : A$ then $stitch_w(shred_{s,p}(A)) = /\!\!/ s \backslash\!\!\backslash_{p,w}$.*

PROOF. By induction on the structure of $A$.

**Case $O$:**

$$\begin{array}{ll}
& stitch_c(shred_{s,p}(O)) \\
= & \\
& c \\
= & \\
& /\!\!/ s \backslash\!\!\backslash_{p,c}
\end{array}$$

**Case $\langle \overrightarrow{\ell = A}\rangle$:**

$$
\begin{aligned}
& \mathit{stitch}_{\langle \ell_i = w_i \rangle_{i=1}^n}(\mathit{shred}_{s,p}(\langle \ell_i : A \rangle_{i=1}^n)) \\
= \quad & \\
& \langle \ell_i = \mathit{stitch}_{w_i}(\mathit{shred}_{s,p.l}(A)) \rangle_{i=1}^n \\
= \quad & \quad \text{(Induction hypothesis)} \\
& \langle \ell_i = /\!\!/ s \backslash\!\!\backslash_{p.l,w_i} \rangle_{i=1}^n \\
= \quad & \\
& /\!\!/ s \backslash\!\!\backslash_{p,\langle \ell_i = w_i \rangle_{i=1}^n}
\end{aligned}
$$

**Case** Bag $A$**:**

$$
\begin{aligned}
& \mathit{stitch}_I(\mathit{shred}_{s,p}(\mathrm{Bag}\, A)) \\
= \quad & \\
& \mathit{stitch}_I(((\mathit{shred}_{s,p.\downarrow}(A))^{\llbracket s \rrbracket_p})) \\
= \quad & \\
& [(\mathit{stitch}_n(\mathit{shred}_{s,p.\downarrow}(A)) @ I_{\mathsf{in}}) \\
& \quad \mid \langle I_{\mathsf{out}}, w \rangle @ I_{\mathsf{in}} \leftarrow \llbracket s \rrbracket_p^\star, I_{\mathsf{out}} = I] \\
= \quad & \quad \text{(Induction hypothesis)} \\
& [/\!\!/ s \backslash\!\!\backslash_{p.\downarrow,w} @ I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, w \rangle @ I_{\mathsf{in}} \leftarrow \llbracket s \rrbracket_p^\star, I_{\mathsf{out}} = I] \\
= \quad & \quad \text{(Lemma 28)} \\
& /\!\!/ s \backslash\!\!\backslash_{p,I}
\end{aligned}
$$

This completes the proof. □

PROOF OF THEOREM 22. By Theorem 29, setting $p = \epsilon$ and $w = \top \diamond 1$. □

We now obtain the main result.

PROOF OF THEOREM 4. Recall the statement of Theorem 4: we need to show that $\vdash L : \mathrm{Bag}\, A$ then:

$$\mathit{stitch}(\mathcal{H}\llbracket L \rrbracket_{\mathrm{Bag}\, A}) = \mathit{stitch}(\mathit{shred}_{\mathcal{A}\llbracket L \rrbracket}(\mathrm{Bag}\, A)) = \mathcal{A}\llbracket L \rrbracket$$

The first equation follows immediately from Theorem 20, and the second from Lemma 21 and Theorem 22.

Furthermore, applying *erase* to both sides we have:

$$\mathit{erase}(\mathit{stitch}(\mathcal{H}\llbracket L \rrbracket_{\mathrm{Bag}\, A})) = \mathit{erase}(\mathcal{A}\llbracket L \rrbracket) = \mathcal{N}\llbracket L \rrbracket$$

where the second step follows by Theorem 19. □

## E. RECORD FLATTENING

*Flat types.* For simplicity we extend base types to include the unit type $\langle\rangle$. This allows us to define an entirely syntax-directed *unflattening* translation from flat record values to nested record values.

$$
\begin{array}{lrl}
\text{Types} & A, B & ::= \mathrm{Bag}\, \langle \overrightarrow{\ell : O} \rangle \\
\text{Base types} & O & ::= \mathit{Int} \mid \mathit{Bool} \mid \mathit{String} \mid \langle\rangle
\end{array}
$$

The Links implementation diverges slightly from the presentation here. Rather than treating the unit type as a base type, it relies on type information to construct units when unflattening values.

*Flat terms.*

$$
\begin{array}{lrl}
\text{Query terms} & L, M & ::= \biguplus \vec{C} \\
\text{Comprehensions} & C & ::= \mathsf{let}\ q = S\ \mathsf{in}\ S' \\
\text{Subqueries} & S & ::= \mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}\ R \\
\text{Data sources} & u & ::= t \mid q \\
\text{Generators} & G & ::= x \leftarrow u \\
\text{Inner terms} & N & ::= X \mid \mathsf{index} \\
\text{Record terms} & R & ::= \langle \overrightarrow{\ell = N} \rangle \\
\text{Base terms} & X & ::= x.\ell \mid c(\vec{X}) \mid \mathsf{empty}\ L
\end{array}
$$

*Flattening types.* The record flattening function $(-)^{\succ}$ flattens record types.

$$
\begin{aligned}
(\mathrm{Bag}\, A)^{\succ} &= \mathrm{Bag}\, (A^{\succ}) \\
O^{\succ} &= \langle \bullet : O \rangle \\
\langle\rangle^{\succ} &= \langle \bullet : \langle\rangle \rangle \\
\langle \overrightarrow{\ell : F} \rangle^{\succ} &= \langle [(\ell_i\_\ell') : O \mid i \leftarrow \mathit{dom}(\vec{\ell}), (\ell' : O) \leftarrow F_i^{\succ}] \rangle
\end{aligned}
$$

Labels in nested records are concatenated with the labels of their ancestors. Base and unit types are lifted to 1-ary records (with a special $\bullet$ field) for uniformity and to aid with reconstruction of nested values from flattened values.

*Flattening terms.* The record flattening function $(-)^{\succ}$ is defined on terms as well as types.

$$
\begin{aligned}
(\biguplus_{i=1}^n C_i)^{\succ} &= \biguplus_{i=1}^n (C_i)^{\succ} \\
(\mathsf{let}\ q = S_{\mathsf{out}}\ \mathsf{in}\ S_{\mathsf{in}})^{\succ} &= \mathsf{let}\ q = {S_{\mathsf{out}}}^{\succ}\ \mathsf{in}\ {S_{\mathsf{in}}}^{\succ} \\
(\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ \mathsf{return}\ N)^{\succ} &= \mathsf{for}\ (\vec{G}\ \mathsf{where}\ X^{\dagger})\ \mathsf{return}\ N^{\succ}
\end{aligned}
$$

$$
\begin{aligned}
X^{\succ} &= \langle \bullet = X^{\dagger} \rangle \\
\langle\rangle^{\succ} &= \langle \bullet = \langle\rangle \rangle \\
(\langle \ell_i = N_i \rangle_{i=1}^m)^{\succ} &= \langle \ell_i\_\ell'_j = X_j \rangle_{(i=1,j=1)}^{(m,n_i)}, \\
& \qquad \mathsf{where}\ {N_i}^{\succ} = \langle \ell'_j = X_j \rangle_{j=1}^{n_i}
\end{aligned}
$$

$$
\begin{aligned}
(x.\ell_1.\cdots.\ell_n)^{\dagger} &= x.\ell_1\_\ldots\_\ell_n\_\bullet \\
c(X_1, \ldots, X_n)^{\dagger} &= c((X_1)^{\dagger}, \ldots, (X_n)^{\dagger}) \\
(\mathsf{empty}\ L)^{\dagger} &= \mathsf{empty}\ L^{\succ}
\end{aligned}
$$

The auxiliary $(-)^{\dagger}$ function flattens $n$-ary projections.

*Type soundness.*

$$\vdash L : A \quad \Rightarrow \quad \vdash L^{\succ} : A^{\succ}$$

*Unflattening record values.*

$$[r_1, \ldots, r_n]^{\prec} = [(r_1)^{\prec}, \ldots, (r_n)^{\prec}]$$

$$\begin{aligned}\langle \bullet = c \rangle^{\prec} &= c \\ \langle \bullet = \langle\rangle \rangle^{\prec} &= \langle\rangle \\ (\langle \ell_i\_\ell'_j = c_j \rangle_{(i=1,j=1)}^{(m,n_i)})^{\prec} &= \langle \ell_i = (\langle \ell'_j = c_j \rangle_{j=1}^{n_i})^{\prec} \rangle_{i=1}^{m}\end{aligned}$$

*Type soundness.*

$$\vdash L : A \quad \Rightarrow \quad \vdash L^{\prec} : A^{\prec}$$

*Correctness*

PROPOSITION 30. *If L is a let-inserted query and* $\vdash L : \mathrm{Bag}\, A$*, then*

$$\left(\mathcal{L}[\![L^{\succ}]\!]\right)^{\prec} = \mathcal{L}[\![L]\!]$$