Shredding higher-order nested queries

Sam Lindley
University of Strathclyde
sam.lindley@strath.ac.uk

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

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

ABSTRACT

We present a modular account of *query shredding*, the simulation of a single nested relational query by a number of flat relational queries, applicable to both set and multiset semantics. Our key insight is that shredding can be greatly simplified by first rewriting the input query into a canonical normal form. Normalisation allows us to define shredding translations on types and terms independently of one another, unlike previous work. An added benefit of normalisation is that we support higher-order terms for free, provided that the result type is a plain nested relation type (without higher order components). In order to generate SQL we consider several alternatives for generating indexes, focusing on a lightweight use of SQL OLAP features.

We prove correctness of our translations, focusing on the central shredding step: shredding a nested query, running the shredded queries, and stitching the results back together yields the same results as running the nested query directly.

1. INTRODUCTION

Databases are arguably one of the most important applications of functional or 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 one type constructor inside another. Motivated by this so-called *impedance mismatch*, and inspired by insights into language design based on monadic comprehensions [28], database researchers introduced nested relational query languages [22, 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 [31] and PigLatin [17], have further extended these ideas, and they have have been particularly influential on language-integrated querying systems such as Kleisli [30], Microsoft's LINQ [15], Links [8], Ferry [10], Ur/Web [6], and SML# [16].

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 functional programming languages, particularly languages such as Links, Ferry, and LINQ that support both nesting and higher-order functions.

Understanding the relative expressiveness of different query languages, such as nested and flat relational query languages, is a classical topic in database theory, with many influential papers [18, 29, 26]. The Flat–Flat Theorem of Paredaens and Van Gucht [18] showed that nested relational queries are no more expressive than flat queries over flat relations. This result was strengthened by Wong [29], whose *Conservativity Theorem* showed that a query over inputs and outputs of nesting degree n does not require building intermediate data structures of greater degree. Wong's proof gave a concrete algorithm for normalising first-order nested relational queries over flat input data to a form that can be easily translated to SQL. Van den Bussche gave an alternative proof of the Flat–Flat Theorem by a simulation technique [26]. Recently, Cooper [7] extended Wong's conservativity result to a higher-order nested relational calculus, in the context of the Links programming language [8]. This required a logical relations argument for collection types, using proof techniques that have only been developed recently; Cooper's proof adapted the approach of Lindley and Stark [14].

Links and LINQ support queries over nested data structures (e.g. records containing nested sets, bags, or lists) in principle; however, 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 (so-called *query storms* [11]). Recently Grust et al. [10, 11] have developed a first-order, nested relational query language called Ferry whose implementation performs shredding via a technique called *loop-lifting*. However, loop-lifting is a monolithic program transformation that is difficult to verify, and produces queries that make heavy use of advanced On-Line Analytic Processing (OLAP) features of SQL:1999 such as `row_number` that are challenging to optimise.

Query shredding has long been known to be possible in principle for nested relational queries over *sets*; this was apparently a folklore result for several years until the publication of Van den Bussche's simulation [26]. To our knowledge, however, Van den Bussche's simulation has never been implemented (nor was it proposed as an efficient implementation technique); it does not appear to be practical for query shredding because of its free use of conversion between the active-domain relational calculus and relational algebra. Moreover, Van den Bussche's simulation does not work for lists or bags; 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 illustrating the problem in Appendix A.

In this paper, we introduce a new, modular approach to query shredding that works uniformly for sets and bags. Our work is oriented towards the programming model of the Links programming language, but should be applicable to other systems based on similar ideas, including Ferry. Essentially, our approach decomposes the translation from nested to flat queries into a number of simpler translations. These compilation passes are much easier to implement, verify, debug, and extend than a monolithic whole-program compilation pass. Our approach, like Ferry, employs some of SQL:1999's OLAP features, but we delay their use as long as possible, so that at most one such operation is needed per generated query, and we identify cases where their use can be avoided.

Our approach works as follows. The first phase, *normalisation*, translates a higher-order query to a *normal form* in which higher-order features have been eliminated. The normalisation phase builds on the rewriting approach taken in a series of papers by Wong [29], Cooper [7], and our prior work [13], but differs in some minor details (given in Appendix C) that simplify the proof of correctness and facilitate later stages. A distinctive feature of our approach is that we support higher-order query constructs in the source language; however, these features are removed in the first stage, which considerably simplifies subsequent stages.

The second phase, *shredding*, 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, additional keys and foreign keys. The shredding algorithm, and its definition and proof of correctness, are new and are the main contributions of the paper. Shredding leverages the normalisation phase in that we can define translations on types and terms independently (in contrast to van den Bussche's or Ferry's approaches).

The third phase, *let-insertion*, hoists nested subqueries using a let-binding construct (equivalent to SQL's with) and a row-numbering operation (equivalent to a restriction of SQL's row_number). Let-insertion is conceptually straightforward, but provides a vital link to proper SQL by providing an implementation of abstract indexes. Finally, translation to SQL requires a final *record flattening* phase; its (standard) details are included in Appendix E for completeness.

1.1 An example

To motivate and illustrate our work, we present a small example showing how our shredding algorithm could be used to provide useful functionality to programmers working in a database programming language such as Ferry or Links. We first describe the code the programmer would actually write and the results the system produces. We then illustrate how the shredding algorithm implements this behaviour efficiently when the nested data is stored in tables on a relational database.

Nested database schemas allow free mixing of collection (bag) types with record or base types. Consider a nested database schema for an organisation consisting of a collection of departments. 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.

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

The nested schema Σ contains a single collection $organisation$ with type $Organisation$.

To illustrate the main features of shredding, we construct 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:

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

In our query language comprehensions are primitive, and we can define and use a higher-order *filter* function to build queries. Likewise, we introduce a higher-order function *get* that returns the name and tasks of each element of a collection.

$$\begin{aligned}
&filter\ p\ xs = \mathsf{for}\ (x \leftarrow xs)\ \mathsf{where}\ (p(x))\ \mathsf{return}\ x\\
&get\ xs\ f = \mathsf{for}\ (x \leftarrow xs)\ \mathsf{return}\ \langle \mathsf{name} = x.\mathsf{name}, \mathsf{tasks} = f\ x\rangle
\end{aligned}$$

The query Q returns each department, the people of interest associated with that department, and their tasks. We assign the special task "buy" to clients.

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

The result type of Q is:

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

The result of running Q is:

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

The nested input data above can be stored in flat relational tables in a number of ways; one way is via the following schema $\Sigma^\flat$ containing four tables:

$$\begin{aligned}
\Sigma^\flat(tasks) &= \mathrm{Bag}\ \langle \mathsf{employee} : String, \mathsf{task} : String\rangle\\
\Sigma^\flat(employees) &= \mathrm{Bag}\ \langle \mathsf{dept} : String, \mathsf{name} : String,\\
&\qquad \mathsf{salary} : Int\rangle\\
\Sigma^\flat(contacts) &= \mathrm{Bag}\ \langle \mathsf{dept} : String, \mathsf{name} : String,\\
&\qquad \mathsf{client} : Bool\rangle\\
\Sigma^\flat(departments) &= \mathrm{Bag}\ \langle \mathsf{name} : String\rangle
\end{aligned}$$

To simplify the example, we assume in addition that every table has an integer-value id field, giving each row a unique identifier; we omit it from the schema for readability. (In the rest of the paper, we will not require that each row has a unique identifier. Instead we provide a feature for generating unique indexes that can ultimately be translated into SQL's row_number construct.) The tabular form of the nested data above is shown in Figure 1.

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

(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 1: Sample data

We can write a query Q_Σ that maps data in the flat schema $\Sigma^\flat$ to the nested schema Σ, defining the nested view *organisation*:

$$
\begin{array}{l}
\mathsf{for}\ (x \leftarrow departments)\\
\quad \mathsf{return}\ (\langle \mathsf{name} = x.\mathsf{name},\\
\qquad\quad \mathsf{employees} =\\
\qquad\qquad \mathsf{for}\ (y \leftarrow employees)\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept})\\
\qquad\qquad\quad \mathsf{return}\ (\langle \mathsf{name} = y.\mathsf{name},\\
\qquad\qquad\qquad\quad \mathsf{tasks} =\\
\qquad\qquad\qquad\qquad \mathsf{for}\ (z \leftarrow tasks)\\
\qquad\qquad\qquad\qquad \mathsf{where}\ (y.\mathsf{name} = z.\mathsf{employee})\\
\qquad\qquad\qquad\qquad\quad \mathsf{return}\ z.\mathsf{task}\\
\qquad\qquad\qquad\quad \mathsf{salary} = y.\mathsf{salary}\rangle)\rangle)\\
\qquad\quad \mathsf{contacts} =\\
\qquad\qquad \mathsf{for}\ (y \leftarrow contacts)\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept})\\
\qquad\qquad\quad \mathsf{return}\ (\langle \mathsf{name} = y.\mathsf{name}, \mathsf{client} = y.\mathsf{client}\rangle)\rangle)
\end{array}
$$

By composing Q with Q_Σ, we obtain a query of type *Result* over input schema $\Sigma^\flat$. Next, using a variant of Cooper's higher-order generalisation of Wong's normalisation algorithm [7, 29], we can normalise the composed query to obtain the following query Q':

$$
\begin{array}{l}
Q' = \mathsf{for}\ (x \leftarrow departments)\\
\qquad\quad \mathsf{return}^a\\
\qquad\quad\ \langle \mathsf{department} = x.\mathsf{name},\\
\qquad\quad\ \ \mathsf{people} =\\
\qquad\qquad (\mathsf{for}\ (y \leftarrow employees)\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \wedge\\
\qquad\qquad\qquad (y.\mathsf{salary} < 1000 \vee y.\mathsf{salary} > 1000000))\\
\qquad\qquad\quad \mathsf{return}^b\ \langle \mathsf{name} = y.\mathsf{name},\\
\qquad\qquad\qquad\quad \mathsf{tasks} = \mathsf{for}\ (z \leftarrow tasks)\\
\qquad\qquad\qquad\qquad\qquad \mathsf{where}\ (z.\mathsf{employee} = y.\mathsf{name})\\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{return}^c\ z.\mathsf{task}\rangle)\\
\qquad\qquad \uplus\\
\qquad\qquad (\mathsf{for}\ (y \leftarrow contacts)\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \wedge y.\mathsf{client})\\
\qquad\qquad\quad \mathsf{return}^d\ \langle \mathsf{name} = y.\mathsf{name},\\
\qquad\qquad\qquad\quad \mathsf{tasks} = \mathsf{return}^e\ \text{“buy”}\rangle)\rangle
\end{array}
$$

Notice that we have annotated each return with a unique tag $a, b, \ldots \in Tag$. These annotations are explained shortly.

Now, however, we are faced with a problem: SQL databases do not directly support nested multisets (or sets). Our shredding algorithm, like Van den Bussche's for sets and Grust et al.'s for lists, can translate a normalised query such as $Q' : Result$ that maps flat input $\Sigma^\flat$ to nested output *Result* to a fixed number of flat queries $Q_1 : Result_1^\flat, \ldots, Q_n : Result_n^\flat$ whose results can be combined via a *stitching* operation $Q^\sharp : Result_1^\flat \times \cdots \times Result_n^\flat \rightarrow Result$. Thus, we can simulate the query Q' by running $Q_1, \ldots, Q_n$ remotely on the database and stitching the results together using $Q^\sharp$. The number of intermediate queries n is the *nesting degree* of *Result*, that is, the number of collection type constructors in the result type. This is not the same as the usual nesting depth; for example, the nesting degree of $\mathrm{Bag}\,\langle A : \mathrm{Bag}\,Int, B : \mathrm{Bag}\,String\rangle$ is 3, not 2. For our example, the nesting degree of *Result* is 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. Let us now consider the top-level query Q_1.

$$
\begin{array}{l}
\mathsf{for}\ (x \leftarrow departments)\\
\quad \mathsf{return}^a\ \langle \mathsf{department} = x.\mathsf{name}, \mathsf{people} = \langle a, x.\mathsf{id}\rangle\rangle
\end{array}
$$

For each iteration, we store an index in place of the nested data. The index consists of a static component a (corresponding to the tag a on the comprehension) and a dynamic component id. The static component identifies which comprehension the index is associated with. The dynamic component identifies the current binding of the comprehension's bound variable. In this case a is redundant because there is only one top-level comprehension, but the need for static indexes will become apparent later. The result of running Q_1 is:

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

Now let us consider the auxiliary query Q_2 for generating the bag bound to the people field.

$$
\begin{array}{l}
\quad (\mathsf{for}\ (x \leftarrow departments)\\
\quad\ \ \mathsf{for}\ (y \leftarrow employees)\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \wedge\\
\qquad\qquad (y.\mathsf{salary} < 1000 \vee y.\mathsf{salary} > 1000000))\\
\qquad \mathsf{return}^b\ (\langle\langle a, x.\mathsf{id}\rangle,\\
\qquad\quad \langle \mathsf{name} = y.\mathsf{name}, \mathsf{tasks} = \langle b, x.\mathsf{id}, y.\mathsf{id}\rangle\rangle\rangle))\\
\uplus\ (\mathsf{for}\ (x \leftarrow departments)\\
\quad\ \ \mathsf{for}\ (y \leftarrow contacts)\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \wedge y.\mathsf{client})\\
\qquad \mathsf{return}^d\ (\langle\langle a, x.\mathsf{id}\rangle,\\
\qquad\quad \langle \mathsf{name} = y.\mathsf{name}, \mathsf{tasks} = \langle d, x.\mathsf{id}, y.\mathsf{id}\rangle\rangle\rangle))
\end{array}
$$

The idea here is that we can ultimately 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.

More interestingly, this query also generates 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. Like Van den Bussche [26], we are using vectors of atomic values (integers) as indexes to identify parts of nested collections.

The result of running Q_2 is:

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

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 innermost query Q_3 for generating the bag bound to the tasks field.

$$\begin{array}{l}(\mathsf{for}\,(x \leftarrow \mathit{departments})\\ \quad \mathsf{for}\,(y \leftarrow \mathit{employees})\;\mathsf{where}\,(x.\mathsf{name} = y.\mathsf{dept} \wedge\\ \qquad\qquad (y.\mathsf{salary} < 1000 \vee y.\mathsf{salary} > 1000000))\\ \quad \mathsf{for}\,(z \leftarrow \mathit{tasks})\;\mathsf{where}\,(z.\mathsf{employee} = y.\mathsf{employee})\\ \qquad \mathsf{return}^c\,\langle\langle b, x.\mathsf{id}, y.\mathsf{id}\rangle, z.\mathsf{task}\rangle)\\ \uplus\,(\mathsf{for}\,(x \leftarrow \mathit{departments})\\ \quad \mathsf{for}\,(y \leftarrow \mathit{contacts})\;\mathsf{where}\,(x.\mathsf{name} = y.\mathsf{dept} \wedge y.\mathsf{client})\\ \qquad \mathsf{return}^e\,\langle\langle d, x.\mathsf{id}, y.\mathsf{id}\rangle, \text{“buy”}\rangle)\end{array}$$

This query does not have any inner indexes because it is at the deepest level of nesting so does not construct any further nested collections. If it did, then we could use the static indexes c and e in the same way as before, but we would run into typing difficulties with the dynamic indexes, as the left branch iterates over three tables, whereas the right branch iterates over two. This limitation is addressed by the more general indexing scheme used in Section 4. The result of running Q_3 is:

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

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 eliminate such nested records from a query with no nested collections in its result type. Thus, we can convert each of Q_1, Q_2, Q_3 to queries that run remotely on any SQL database. For instance, Q_3 can be converted to the SQL:

$$\begin{array}{l}(\mathsf{select}\;2\;\mathsf{as}\;\mathsf{i1\_1}, x.\mathsf{id}\;\mathsf{as}\;\mathsf{i1\_2}, y.\mathsf{id}\;\mathsf{as}\;\mathsf{i1\_3}, z.\mathsf{task}\;\mathsf{as}\;\mathsf{i2}\\ \;\mathsf{from}\;\mathit{departments}\;\mathsf{as}\;x, \mathit{employees}\;\mathsf{as}\;y, \mathit{tasks}\;\mathsf{as}\;z\\ \;\mathsf{where}\quad ((x.\mathsf{name} = y.\mathsf{dept} \wedge\\ \qquad\qquad (y.\mathsf{salary} < 1000 \vee y.\mathsf{salary} > 1000000))\\ \qquad\quad \wedge\,(x.\mathsf{employee} = y.\mathsf{employee}))\\ \;\mathsf{union\;all}\\ (\mathsf{select}\;4\;\mathsf{as}\;\mathsf{i1\_1}, x.\mathsf{id}\;\mathsf{as}\;\mathsf{i1\_2}, y.\mathsf{id}\;\mathsf{as}\;\mathsf{i1\_3}, \text{“buy”}\;\mathsf{as}\;\mathsf{i2}\\ \;\mathsf{from}\;\mathit{departments}\;\mathsf{as}\;x, \mathit{contacts}\;\mathsf{as}\;y\\ \;\mathsf{where}\,(x.\mathsf{name} = y.\mathsf{dept} \wedge y.\mathsf{client})\end{array}$$

Note that each static index $a, b, c, \ldots$ has been converted to a number $1, 2, 3, \ldots$, and records have been flattened by concatenating field names using ␣ as a delimiter.

The three queries have the following result types:

$$\begin{array}{l}\mathit{Result}_1 = \mathrm{Bag}\,\langle \mathsf{department} : \mathit{String}, \mathsf{people} : \mathit{Index}\rangle\\ \mathit{Result}_2 = \mathrm{Bag}\,\langle \mathit{Index}, \langle \mathsf{name} : \mathit{String}, \mathsf{tasks} : \mathit{Index}'\rangle\rangle\\ \mathit{Result}_3 = \mathrm{Bag}\,\langle \mathit{Index}', \mathit{String}\rangle\end{array}$$

where $\mathit{Index} = \langle \mathit{Tag}, \mathit{Int}\rangle$ and $\mathit{Index}' = \langle \mathit{Tag}, \mathit{Int}, \mathit{Int}\rangle$. Once the results $R_1 : \mathit{Result}_1, R_2 : \mathit{Result}_2, R_3 : \mathit{Result}_3$ 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 follows the same idea as the query Q_Σ that constructs the nested version of Σ from $\Sigma^\flat$.

$$\begin{array}{l}\mathsf{for}\,(x \leftarrow Q_1)\\ \;\mathsf{return}\,(\langle \mathsf{department} = x.\mathsf{name},\\ \qquad\quad \mathsf{people} =\\ \qquad\qquad \mathsf{for}\,(\langle i, y\rangle \leftarrow Q_2)\;\mathsf{where}\,(x.\mathsf{people} = i))\\ \qquad\qquad\; \mathsf{return}\,(\langle \mathsf{name} = y.\mathsf{name},\\ \qquad\qquad\qquad\quad \mathsf{tasks} =\\ \qquad\qquad\qquad\qquad \mathsf{for}\,(\langle j, z\rangle \leftarrow Q_3)\;\mathsf{where}\,(y.\mathsf{tasks} = j)\\ \qquad\qquad\qquad\qquad\; \mathsf{return}\;z\rangle)\rangle)\end{array}$$

We want to emphasise, however, that all of the code mentioned in this discussion is generated by the *implementation*, not by the programmer. All the programmer needs to do is write the concise, higher-order, nested query shown earlier.

We also want to emphasise that it is unnecessary to construct the nested version of the input data explicitly in-memory or on the database. Although the notional first step of our implementation approach is to apply query Q_Σ that constructs the nested database from the flat tables, there is no need to materialise this data structure anywhere. Instead, by composing with Q and normalising, we effectively deforest the intermediate nested data structure. In the common case where the query result is much smaller than the full database, this is usually much faster than shipping all of the data to the host system.

In the rest of the paper we focus attention on the critical step of transforming a query Q whose input is flat and output is nested to multiple flat queries $Q_1, \ldots, Q_n$, such that evaluating Q (in-memory) is equivalent to evaluating $Q_1, \ldots, Q_n$ on the database and then stitching the results together on the host system.

1.2 Contributions

In contrast to Van den Bussche's simulation, our approach handles higher-order features and compiles to SQL. In contrast to Ferry, we do not just describe an algorithm but provide a proof of correctness. Our approach does have some limitations, which are the subject of current work: at present, it does not recognise idiomatic SQL-style grouping and aggregation, nor does it handle queries that return functions. Some work has been done to handle both features in Ferry, albeit without any correctness proof [25].

Also, it is important to note that Ferry's approach is geared towards a list-based interpretation of database queries, while we assume a bag-based semantics (matching proper SQL), and set-based semantics can be accommodated simply by eliminating duplicates in the final result. In fact, the core technical part of the paper (Sections 4–6) works just as well for a list semantics. The only parts that rely on unordered set or bag semantics are normalisation (Section 3) and conversion to SQL (Section 7). We leave extensions to handle list semantics to future work.

The rest of the paper is organised as follows. Section 2 gives background and Section 3 reviews query normalisation. Section 4 defines a translation from normal forms to shredded terms, *shredded packages* for bundling shredded queries or shredded results together, and a semantics for shredded query packages Section 5 summarises the proof of correctness of the shredding translation. Section 6 gives a translation for providing flat indexes using let-insertion. Section 7 outlines a translation to SQL. Section 8 discusses our prototype implementation. Section 9 discusses related work and Section 10 concludes.

The details of proofs are given in appendices. Also, for completeness, the details of the initial normalisation stage and the (standard) record-flattening stage are given in appendices.

2. BACKGROUND

2.1 Notational conventions

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

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

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.

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 $[v \mid 1 \leq i \leq n]$ and similarly, e.g., $\langle \ell_i = M_i\rangle_{i=1}^n$ for $\langle \ell_1 = M_1, \ldots, \ell_n = M_n\rangle$.

2.2 Nested relational calculus

We take the higher-order, nested relational calculus (evaluated over bags) as our starting point. We call this λ_{NRC}. The types of λ_{NRC} include base types (integers, strings, booleans), record types $\langle\overrightarrow{\ell : A}\rangle$, bag types Bag A, and function types $A \to B$.

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

We say that a type is *nested* if it contains no function types and *flat* if it is positive and contains no collection types.

The terms of λ_{NRC} include λ-abstractions, applications, and the standard terms of nested relational calculus.

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

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 $\#_1, \#_2, \ldots$ and encode tuple types $\langle A_1, \ldots, A_n\rangle$ as record types $\langle \#_1 : A_1, \ldots, \#_n : A_n\rangle$, 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 former are constrained to have flat relation type (Bag $\langle \ell_1 : O_1, \ldots, \ell_n : O_n\rangle$), and the latter to be 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. (That is, it is a monadic bind or concat-map operation, not a map). The expression $\mathsf{empty}\ M$ evaluates to true if M evaluates to an empty bag, and false otherwise.

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 write $v \simeq v'$ if v and v' are equivalent up to permutation of list elements. The reason for using lists rather than bags here is that

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

Figure 2: Semantics of higher-order nested relational queries

the order in which indexes are generated must be consistent across shredded queries.

We interpret base types as base types, functions as functions, records as records, and bags as lists. For each table $t \in \mathit{dom}(\Sigma)$, 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. The most general approach, which we adopt, is to order by all of the columns arranged in lexicographic order (assuming some ordering on field names).

We assume fixed interpretations $[\![c]\!]$ for the constants and primitive operations. The semantics of nested relational calculus are shown in Figure 2; and the (standard) typing rules are given in Appendix 8; our previous paper [13] shows how to embed λ_{NRC} into a general-purpose programming language using a richer *type-and-effect* system.

If $\Gamma \vdash M, N : A$ then we write $M \simeq N$ when $\mathcal{N}[\![M]\!]_\rho \simeq \mathcal{N}[\![N]\!]_\rho$ holds for all $\rho : \Gamma$. We let ρ range over environments mapping variables to values, writing ε for the empty environment and $\rho[x \mapsto v]$ for the extension of ρ with x bound to v.

3. QUERY NORMALISATION

The first stage of our shredding algorithm is to rewrite the query to a normal form that has a number of useful properties. This is a simple adaptation of normalisation techniques in prior work [29, 7, 13]. Essentially, the idea is to apply η-expansion, β-reduction, and commuting conversions for bag comprehensions to reduce the query to a form with no λ-abstractions and no superfluous comprehensions (see also [14, 12]). The full details are given in Appendix C.

In Links, query normalisation is an important part of the execution model [7, 13]. When a subexpression denoting a flat–flat 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 execution proceeds. Note that at run-time, the query expression is *closed* in the sense that all of its free variables have been replaced with values; however, it may still contain table names whose values are stored remotely in the database.

For *flat–nested* queries that read from flat tables and produce a nested result value, our normalisation procedure is similar, but in contrast to Cooper's algorithm, we hoist all conditionals into the nearest enclosing comprehension as where clauses. The resulting normal forms are:

$$
\begin{array}{lrl}
\text{Query terms} & L ::= & \biguplus \vec{C} \\[4pt]
\text{Comprehensions} & C ::= & \mathsf{for}\,(\vec{G}\,\mathsf{where}\,X)\,\mathsf{return}\,M \\
\text{Generators} & G ::= & x \leftarrow t \\
\text{Normalised terms} & M, N ::= & X \mid R \mid L \\
\text{Record terms} & R ::= & \langle \overrightarrow{\ell = M} \rangle \\
\text{Base terms} & X ::= & x.\ell \mid c(\vec{X}) \mid \mathsf{empty}\,L
\end{array}
$$

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

THEOREM 1. *Given a closed flat–nested query* $\vdash M : \mathrm{Bag}\,A$*, there exists a normalisation function* $norm_{\mathrm{Bag}\,A}$*, such that* $M \simeq norm_{\mathrm{Bag}\,A}(M)$ *is in normal form.*

The normalisation algorithm and correctness proof are similar to those in previous papers [7, 13], and are given in Appendix C. The normal forms for flat–nested queries do not correspond directly to SQL; the next two sections show how to bridge this gap.

4. SHREDDING

4.1 The shredding translation

In order to shred nested queries, we introduce an abstract type *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 identifies which comprehension the index is associated with. The dynamic component identifies the current binding of the variable x in the comprehension.

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:

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

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:

$$
\begin{array}{lrl}
\text{Query terms} & L, M ::= & \biguplus \vec{C} \\[4pt]
\text{Comprehensions} & C ::= & \mathsf{return}^a\,\langle I, N \rangle \\
 & \mid & \mathsf{for}\,(\vec{G}\,\mathsf{where}\,X)\,C \\
\text{Generators} & G ::= & x \leftarrow t \\
\text{Inner terms} & N ::= & X \mid R \mid I \\
\text{Record terms} & R ::= & \langle \overrightarrow{\ell = N} \rangle \\
\text{Base terms} & X ::= & x.\ell \mid c(\vec{X}) \mid \mathsf{empty}\,L \\[8pt]
\text{Indexes} & I, J ::= & a \diamond d \\
\text{Dynamic indexes} & d ::= & \mathsf{out} \mid \mathsf{in}
\end{array}
$$

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. Inner terms 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 always appears in the left component of a comprehension body, and only there, 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.

An additional constraint that is not captured by the above grammar is that all comprehensions C_i in a shredded query $\biguplus_{i=1}^{n} C_i$ must have the same non-zero *nesting degree*, that is, there exists $m > 0$ such that $degree(C_i) = m$ for $1 \le i \le n$, where:

$$
\begin{aligned}
degree(\mathsf{return}^a\,\langle I, N \rangle) &= 0 \\
degree(\mathsf{for}\,(\vec{G}\,\mathsf{where}\,X)\,C) &= 1 + degree(C)
\end{aligned}
$$

This constraint will be guaranteed by the shredding transformation.

To define the shredding transformation we use paths to point to parts of types. We use paths made up of the symbol $\downarrow$ (representing traversing a bag constructor) and record labels ℓ.

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

As paths are simply lists of $\downarrow$s and labels, we will sometimes write $p.{\downarrow}$ for the path p with $\downarrow$ appended at the end and similarly for $p.\ell$. Furthermore, we will write $p.\vec{\ell}$ for the path p with all the labels of $\vec{\ell}$ appended. The function $paths(A)$ defines the set of paths to bags in a type A:

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

The *nesting degree* of a type A is the number of paths to bag constructors in A, that is, $degree(A) = |paths(A)|$.

We now define a shredding translation on types. Given a path $p \in paths(A)$, the type $\llbracket A \rrbracket_p$ is the *outer shredding* of A at p, a shredded type that corresponds to the bag at path p in A. This is defined in terms of the *inner shredding* $\llfloor A \rrfloor$, a flat type that represents the contents of the bag.

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

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

For example, consider the example result type *Result* from Section 1.1. Its nesting degree is 3, and its paths are:

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

Moreover, we can shred it in three ways using these three paths:

$$
\begin{aligned}
\llbracket \mathit{Result} \rrbracket_{\epsilon} &= \mathrm{Bag}\,\langle \mathit{Index}, \langle \mathsf{department} : \mathit{String}, \mathsf{people} : \mathit{Index} \rangle \rangle \\
\llbracket \mathit{Result} \rrbracket_{\downarrow.\mathsf{people}.\epsilon} &= \mathrm{Bag}\,\langle \mathit{Index}, \langle \mathsf{name} : \mathit{String}, \mathsf{tasks} : \mathit{Index} \rangle \rangle \\
\llbracket \mathit{Result} \rrbracket_{\downarrow.\mathsf{people}.\downarrow.\mathsf{tasks}.\epsilon} &= \mathrm{Bag}\,\langle \mathit{Index}, \mathit{String} \rangle
\end{aligned}
$$

The shredding translation on terms $\llbracket L \rrbracket_p$ is given in Figure 3. 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 $\llfloor M \rrfloor_a$ produces a flat representation of M with inner static index

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

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

Figure 3: Shredding translation on terms

a. Note that the shredding translation is linear in time and space. Observe that for emptiness tests we need only the top-level query.

As a basic sanity check, we show that well-formed normalised terms shred to well-formed shredded terms of the appropriate shredded types. We will show the full correctness 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 $\vdash \lceil\!\lceil L \rceil\!\rceil_p : \lceil\!\lceil A \rceil\!\rceil_p$.*

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.

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

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 write Λ_S for the set of shredded terms, and $\mathcal{T}_S$ for the set of shredded types. We sometimes omit the type parameter when it is clear from context. Typing rules for shredded packages are shown in Appendix 8.

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 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 $\mathit{erase}(\hat{A}) = \mathit{erase}(\hat{A}')$.

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

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

$$\mathit{package}_f(A) = \mathit{package}_{f,\epsilon}(A)$$

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

Using *package*, we lift the type-level and term-level shredding functions $\lceil\!\lceil - \rceil\!\rceil_{-}$ 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.

$$
\begin{aligned}
\mathit{shred}_B(A) &= \mathit{package}_{(\lceil\!\lceil B \rceil\!\rceil_{-})}(A) \\
\mathit{shred}_L(A) &= \mathit{package}_{(\lceil\!\lceil L \rceil\!\rceil_{-})}(A)
\end{aligned}
$$

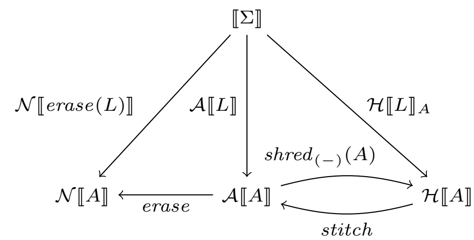

Figure 4: Correctness of shredding and stitching

For example, shredding *Result* type from the introduction gives:

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

Shredding the query Q' gives the same package, except the type annotations A_1, A_2, A_3 become queries Q_1, Q_2, Q_3. We will use Q_2 as a running example for the rest of the paper.

$$
\begin{aligned}
Q_2 = \quad &(\mathsf{for}\,(x \leftarrow \mathit{departments}) \\
&\ \ \mathsf{for}\,(y \leftarrow \mathit{employees})\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \wedge \\
&\qquad\qquad (y.\mathsf{salary} < 1000 \vee y.\mathsf{salary} > 1000000)) \\
&\ \ \ \mathsf{return}^{b}\,(\langle a \diamond \mathsf{out}, \langle \mathsf{name} = y.\mathsf{name}, \mathsf{tasks} = b \diamond \mathsf{in} \rangle \rangle)) \\
\uplus\ &(\mathsf{for}\,(x \leftarrow \mathit{departments}) \\
&\ \ \mathsf{for}\,(y \leftarrow \mathit{contacts})\ \mathsf{where}\ (x.\mathsf{name} = y.\mathsf{dept} \wedge y.\mathsf{client}) \\
&\ \ \ \mathsf{return}^{d}\,(\langle a \diamond \mathsf{out}, \langle \mathsf{name} = y.\mathsf{name}, \mathsf{tasks} = d \diamond \mathsf{in} \rangle \rangle))
\end{aligned}
$$

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 $\mathit{erase}(\mathit{shred}_A(A)) = A$. Furthermore, if L is a closed, normalised, flat–nested query such that $\vdash L : A$ then $\vdash \mathit{shred}_L(A) : \mathit{shred}_A(A)$ also.*

5. CORRECTNESS

The main result we wish to prove 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: a) define shredding on nested values; b) show that shredding commutes with query execution; and c) show that stitching is a left-inverse to shredding of values.

Unfortunately this naive proof approach fails, 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 4.1. 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 4, where we view a query as a function from the interpretation of the Σ to the interpretation of its result type. To prove correctness is to prove that this diagram commutes.

5.1 Annotated semantics over nested values

We annotate bag elements with distinct indexes as follows:

$$\begin{array}{ll} \text{Results} & s ::= [v_1@I_1, \dots, v_m@I_m] \\ \text{Inner values} & w ::= c \mid r \mid I \\ \text{Rows} & r ::= \langle \ell_1 = w_1, \dots, \ell_n = w_n \rangle \\ \text{Indexes} & I, J \\ \text{Dynamic indexes} & \iota ::= 1 \mid \iota.i \end{array}$$

Indexes themselves are still abstract, but we will construct them from a concrete representation of static and dynamic indexes. Concretely, we represent each dynamic index ι as a list of integers. Each integer identifies the bag element to which the comprehension's variable is bound. The advantage of this particular representation is that it admits a simple declarative semantics, which will make it straightforward to relate the nested semantics with a suitable semantics for shredded queries. The top level dynamic index is always 1.

The *canonical* choice for representing indexes is to take $I = a \diamond \iota$. We also admit alternative indexing schemes, by parameterising the semantics over an indexing function $index$ mapping canonical indexes to a concrete representation. In the simplest case, we take $index$ to be the identity function. More generally, it may 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 in Section 5.5. For now we assume canonical indexes.

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

$$\mathcal{A}[\![L]\!] = \mathcal{A}[\![L]\!]_{\varepsilon,1}$$

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

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

As well as an environment, the current dynamic index is threaded through the semantics.

The $enum$ function takes a list of elements and returns the same list with the element number paired up with each source element.

$$enum([v_1, \dots, v_m]) = [\langle 1, v_1 \rangle, \dots \langle m, v_m \rangle]$$

The index annotations $@I$ on collection elements are needed solely for our correctness proof, and do not need to be materialised at run time.

5.2 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 Appendix B.

$$\text{Results} \quad s ::= [\langle I_1, w_1 \rangle @J_1, \dots, \langle I_m, w_m \rangle @J_m]$$

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

$$[\![s]\!]_p = [\![s]\!]^{\star}_{\top \diamond 1, p}$$

$$\begin{aligned} [\![[v_i@J_i]_{i=1}^{n}]\!]^{\star}_{I,\epsilon} &= [\langle I, \lfloor\!\lfloor v_i \rfloor\!\rfloor_{J_i} \rangle @J_i]_{i=1}^{n} \\ [\![[v_i@J_i]_{i=1}^{n}]\!]^{\star}_{I,\downarrow.p} &= concat([[\![v_i]\!]^{\star}_{J_i,p}]_{i=1}^{n}) \\ [\![\langle \ell_i = v_i \rangle_{i=1}^{n}]\!]^{\star}_{I,\ell_i.p} &= [\![v_i]\!]^{\star}_{I,p} \end{aligned}$$

$$\begin{aligned} \lfloor\!\lfloor c \rfloor\!\rfloor_I &= c \\ \lfloor\!\lfloor \langle \ell_i = v_i \rangle_{i=1}^{n} \rfloor\!\rfloor_I &= \langle \ell_i = \lfloor\!\lfloor v_i \rfloor\!\rfloor_I \rangle_{i=1}^{n} \\ \lfloor\!\lfloor s \rfloor\!\rfloor_I &= I \end{aligned}$$

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 nested types and nested queries.

$$\begin{aligned} shred_s(\text{Bag } A) &= package_{([\![s]\!]_{-})}(\text{Bag } A) \\ shred_{s,p}(\text{Bag } A) &= package_{([\![s]\!]_{-}),p}(\text{Bag } A) \end{aligned}$$

Stitching values. A shredded value package can be stitched back together into a nested value as follows.

$$stitch(\hat{A}) = stitch_{\top \diamond 1}(\hat{A})$$

$$\begin{aligned} stitch_c(O) &= c \\ stitch_{\langle \ell_i = w_i \rangle_{i=1}^{n}}(\langle \ell_i : \hat{A}_i \rangle_{i=1}^{n}) &= \langle \ell_i = stitch_{w_i}(\hat{A}_i) \rangle_{i=1}^{n} \\ stitch_I((\text{Bag } \hat{A})^s) &= [(stitch_w(\hat{A}))@J \\ &\qquad \mid \langle I', w \rangle @J \leftarrow s, I' = I] \end{aligned}$$

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

5.3 Annotated semantics over shredded values

We now show how to interpret shredded queries and query packages over shredded values. The semantics of shredded queries is given in Figure 5. 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 $pmap_{\mathcal{S}[\![-]\!]}\ shred_L(A)$.

For example, here is the shredded package that we obtain after running the query Q':

$$\begin{aligned} \mathcal{H}[\![Q']\!]_A = \text{Bag } \langle &\mathsf{department} : String, \\ &\mathsf{people} : \text{Bag } \langle \mathsf{name} : String, \\ &\qquad \mathsf{tasks} : (\text{Bag } String)^{s_3} \rangle^{s_2} \rangle^{s_1} \end{aligned}$$

where:

$$\begin{aligned} s_1 = [&\langle \top \diamond 1, \langle \mathsf{department} = \text{“Product”}, \mathsf{people} = a \diamond 1.1 \rangle \rangle, \\ &\langle \top \diamond 1, \langle \mathsf{department} = \text{“Quality”}, \mathsf{people} = a \diamond 1.2 \rangle \rangle, \\ &\langle \top \diamond 1, \langle \mathsf{department} = \text{“Research”}, \mathsf{people} = a \diamond 1.3 \rangle \rangle, \\ &\langle \top \diamond 1, \langle \mathsf{department} = \text{“Sales”}, \mathsf{people} = a \diamond 1.4 \rangle \rangle] \\ s_2 = [&\langle a \diamond 1.1, \langle \mathsf{name} = \text{“Bert”}, \mathsf{tasks} = b \diamond 1.1.2 \rangle \rangle, \\ &\langle a \diamond 1.4, \langle \mathsf{name} = \text{“Erik”}, \mathsf{tasks} = b \diamond 1.4.5 \rangle \rangle, \dots] \\ s_3 = [&\langle b \diamond 1.1.2, \text{“build”} \rangle, \langle b \diamond 1.4.5, \text{“call”} \rangle, \dots] \end{aligned}$$

$$\mathcal{S}[\![L]\!] = \mathcal{S}[\![L]\!]_{\varepsilon,1} \qquad \mathcal{S}[\![\langle \ell = N\rangle_{i=1}^n]\!]_{\rho,\iota} = \langle \ell_i = \mathcal{S}[\![N_i]\!]_{\rho,\iota}\rangle_{i=1}^n \qquad \mathcal{S}[\![a \diamond \mathsf{out}]\!]_{\rho,\iota.i} = index(a \diamond \iota)$$

$$\mathcal{S}[\![X]\!]_{\rho,\iota} = \mathcal{N}[\![X]\!]_{\rho} \qquad \mathcal{S}[\![a \diamond \mathsf{in}]\!]_{\rho,\iota.i} = index(a \diamond \iota.i)$$

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

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

Figure 5: Semantics of shredded queries

5.4 Main results

Due to space limits, we relegate the full details of the proof to Appendix D. There are three key theorems. The first states that shredding commutes with the annotated semantics.

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

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

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 $indexes_p(v)$, the indexes of nested value v along path p as follows.

$$indexes_{\epsilon}([v_i @ J_i]_{i=1}^n) = [J_i]_{i=1}^n$$
$$indexes_{\downarrow.p}([v_i @ J_i]_{i=1}^n) = concat([indexes_p(v_i)]_{i=1}^n)$$
$$indexes_{\ell_i.p}(\langle \ell_i = v_i\rangle_{i=1}^n) = indexes_p(v_i)$$

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

LEMMA 5. *If* $\vdash L : A$*, then* $\mathcal{A}[\![L]\!]$ *is well-indexed at type* A*.*

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

THEOREM 6. *If* $\vdash s : \mathrm{Bag}\, A$ *and* s *is well-indexed at type* $\mathrm{Bag}\, A$ *then* $stitch(shred_s(\mathrm{Bag}\, A)) = s$*.*

Combining Theorem 4, Lemma 5 and Theorem 6 we obtain the main correctness result (see Figure 4).

THEOREM 7. *If* $\vdash L : \mathrm{Bag}\, A$ *then:* $stitch(\mathcal{H}[\![L]\!]_{\mathrm{Bag}\,A}) = \mathcal{A}[\![L]\!]$*, and in particular:* $erase(stitch(\mathcal{H}[\![L]\!]_{\mathrm{Bag}\,A})) = \mathcal{N}[\![L]\!]$*.*

5.5 Alternative indexing schemes

In order to formalise *valid* indexing schemes, we first define a function for computing the canonical indexes of a nested query.

$$\mathcal{I}[\![L]\!] = \mathcal{I}[\![L]\!]_{\varepsilon,1}$$

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

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

Note that $\mathcal{I}[\![-]\!]$ follows essentially the same form as $\mathcal{A}[\![-]\!]$, but instead of the nested value v it computes all indexes of v.

We define alternative indexing schemes by instantiating the $index$ parameter of the shredded semantics (see Section 5.3). An $index$ parameter is *valid* with respect to the closed nested query L if it is injective and defined on every canonical index in $\mathcal{I}[\![L]\!]$.

LEMMA 8. *If* $index$ *is valid for* L *then* $\mathcal{A}[\![L]\!]$ *is well-indexed.*

The only requirement on indexes in the proof of Theorem 7 is that nested values be well-indexed, hence the proof extends to any valid indexing scheme. We briefly mention two alternative valid indexing schemes: *natural indexes* and *flat indexes*.

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 [26]. However, for bags this is not always possible, so using natural indexes may require adding extra key fields. The type of natural indexes is a sum or variant type. The static index is a tag, and its payload is a tuple of primary keys.

Given a table t, let key_t be the function that given a row r of t returns the primary key of r. We now define a function to compute the list of natural indexes for a query L.

$$\mathcal{I}^\natural[\![L]\!] = \mathcal{I}^\natural[\![L]\!]_{\varepsilon,1}$$

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

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

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

An advantage of natural indexes is that when translating to SQL it is unnecessary to use the row_number function. Consequently, 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, which is determined by the tables being queried, may vary across the component comprehensions of a shredded query. Variant types can be used to fix this problem.

There are two ways of encoding variants in SQL. The first encoding packs variants into a record. The record consists of a distinguished tag column, and one column for each column of each shredded variant argument in the variant type. The tag column holds an integer encoding which variant tag is active and the columns associated with the encoded tag are populated with their values. The other columns can contain nulls or any other values. The second encoding generates multiple queries: one for each tag.

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

We elaborate on flat indexes in the next two sections, illustrating how they translate to SQL. This depends on the row_number operator, which we use to implement the functionality of *enum*. We choose to focus on flat indexes in the remainder of the paper because they are more generally applicable than natural indexes, which require all rows of the source tables to be distinct.

6. FLAT INDEXES AND LET-INSERTION

Our semantics for shredded queries uses canonical indexes. We now specify an object language providing flat indexes and move closer towards an SQL implementation. 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 iteration.

Following let-insertion, the types are as before, except indexes are represented as pairs of integers.

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

An index is a pair of a static index and a dynamic index; both are simple integers.

The syntax of terms is adapted as follows:

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

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 6. Each comprehension is rearranged into two sub-queries. The first generates the outer indexes. The second computes the results.

For example, applying $\mathbf{L}$ to Q_2 from Section 4.2 yields:

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

The translation sometimes produces n-ary projections in order to refer to values bound by the first subquery inside the second.

To state and prove the correctness of let-insertion, we use the same values as before, but the $@I$ index components are no longer needed. The semantics is given in Figure 7. Rather than maintaining a canonical index, it generates a flat index for each subquery.

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

THEOREM 9. *Given shredded query* $\vdash L : \mathrm{Bag}\,\langle \mathit{Index}, F\rangle$, *then* $\vdash \mathbf{L}(L) : \mathrm{Bag}\,\langle\langle \mathit{Int}, \mathit{Int}\rangle, F'\rangle$ *(where* F' *is the result of replacing* Index *with* $\langle \mathit{Int}, \mathit{Int}\rangle$ *in* F*).*

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, note that the translation does not depend on $\mathit{index}^{\flat}_L$; it is just used to clarify which valid indexing scheme is being used in the semantics $\mathcal{S}$ to prove correctness.

THEOREM 10. *Let* $\vdash L : A$ *and* $\llbracket L \rrbracket_p = M$, *and set* $\mathit{index} = \mathit{index}^{\flat}_L$, *then* $\mathit{erase}(\mathcal{S}\llbracket M \rrbracket) = \mathcal{L}\llbracket \mathbf{L}(M) \rrbracket$.

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 definitions collapse to the same thing if we replace all dynamic indexes by unit. 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_{out} and S_{in}. The dynamic indexes computed by S_{out} coincide exactly with those of $\mathcal{I}^{\flat}\llbracket L \rrbracket$ at static index a, and the dynamic indexes computed by S_{in}, if there are any, coincide exactly with those of $\mathcal{I}^{\flat}\llbracket L \rrbracket$ at static index b. □

7. CONVERSION TO SQL

SQL does not support nested records. Our final translation flattens records. (In fact, we could have performed flattening at any earlier point, but leave it until the end for convenience so that we can use nested records in the earlier phases.) The (standard) details are presented in Appendix E for completeness.

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

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

This fragment of SQL is almost isomorphic to the language we get if we apply let-insertion and record flattening to shredded queries. 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:

$$\mathsf{for}\ (\overrightarrow{x \leftarrow t}\ \mathsf{where}\ X)\ \mathsf{return}\ R$$

is simulated by $\mathsf{row\_number}()\ \mathsf{over}\ (\mathsf{order\ by}\ \overrightarrow{x.\ell})$, where:

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

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.

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

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

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

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

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

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

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

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

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

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

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

$$\mathit{expand}(x, t) = \langle \ell_i = x.\ell_i\rangle_{i=1}^n \text{ where } \Sigma(t) = \mathrm{Bag}\ \langle \overrightarrow{\ell : A}\rangle \qquad \mathit{init}\ [x_i]_{i=1}^n = [x_i]_{i=1}^{n-1} \qquad \mathit{last}\ [x_i]_{i=1}^n = x_n$$

$$\mathit{gens}\ (\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ C) = \vec{G} :: \mathit{gens}\ C \qquad \mathit{conds}\ (\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ C) = X :: \mathit{conds}\ C \qquad \mathit{body}\ (\mathsf{for}\ (\vec{G}\ \mathsf{where}\ X)\ C) = \mathit{body}\ C$$

$$\mathit{gens}\ (\mathsf{return}^a\ N) = [\,] \qquad \mathit{conds}\ (\mathsf{return}^a\ N) = [\,] \qquad \mathit{body}\ (\mathsf{return}^a\ N) = N$$

Figure 6: The let-insertion translation

$$\mathcal{L}[\![L]\!] = \mathcal{L}[\![L]\!]_\varepsilon \qquad \mathcal{L}[\![\biguplus_{j=1}^m C_j]\!]_\rho = \mathit{concat}([\mathcal{L}[\![C_j]\!]_\rho]_{j=1}^m)$$

$$\mathcal{L}[\![t]\!]_\rho = [\![t]\!] \qquad \mathcal{L}[\![q]\!]_\rho = \rho(q)$$

$$\mathcal{L}[\![\langle \ell_j = N_j\rangle_{j=1}^m]\!]_{\rho,i} = \langle \ell_j = \mathcal{L}[\![N_j]\!]_{\rho,i}\rangle_{j=1}^m \qquad \mathcal{L}[\![X]\!]_{\rho,i} = \mathcal{N}[\![X]\!]_\rho \qquad \mathcal{L}[\![\mathsf{index}]\!]_{\rho,i} = i$$

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

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

Figure 7: Semantics of let-inserted shredded queries

Continuing our example, Q_2 becomes:

```
(with q as (select x.name as i1␣name,
                   row_number() over (order by x.name) as i2
            from departments as x)
 select a as i1␣1, z.i2 as i1␣2, y.name as i2␣name, b as i2␣tasks␣1,
        row_number() over
         (order by z.i1␣name, z.i2, y.dept, y.employee, y.salary)
                                                       as i2␣tasks␣2
 from employees as y, q as z
 where (z.i1␣name = y.dept ∧
                    (y.salary < 1000 ∨ y.salary > 1000000)))
union all
(with q as (select x.name as i1␣name,
                   row_number() over (order by x.name) as i2
            from departments as x)
 select a as i1␣1, z.i2 as i1␣2, y.name as i2␣name, d as i2␣tasks␣1,
        row_number() over
         (order by z.i1␣name, z.i2, y.dept, y.name, y.client)
                                                       as i2␣tasks␣2
 from contacts as y, q as z
 where (z.i1␣name = y.dept ∧ y.client))
```

8. IMPLEMENTATION

We believe that our shredding and let-insertion algorithms and their proofs of correctness are significant contributions; however, as motivation, we also argued that our approach is relatively straightforward to implement and should compare favourably to Ferry in terms of performance by avoiding heavy reliance on SQL:1999 OLAP features. To evaluate these claims, we have implemented our approach in Links and compared with an implementation of Ferry in Links due to Ulrich [25]. We should first note that Ulrich's system supports features, such as grouping, that are not handled by our algorithm; thus, we focused on queries that both systems can handle. Also, Ferry is based on list semantics, which may affect its performance by limiting optimisation opportunities. Thus, the experimental results only compare the capabilities of Ferry and our approach viewed as an implementations of multiset semantics.

Query performance. We measured the query execution time for four nested queries on randomly generated data, both for Links extended with our shredding algorithm and for Links+Ferry. All tests were performed using PostgreSQL 9.05 running on a Dell Optiplex 745 with 3Ghz CPU and 4GB of RAM. Full details of the queries are given in Appendix F. T_1 is a plain query that assembles a nested bag of bags of integers from three tables using two nested joins. T_2 is similar, but includes a union operation, joining to two different tables at the deepest level of nesting. T_3 involves a union operation and nested records. T_4 includes an emptiness test and a union operation.

For each query, most of the data occurs at the deepest level of nesting, and the innermost query dominates execution time (both for our algorithm and for the Ferry algorithm). Thus we only report the execution time of the innermost queries. For each measurement, we took the average of five runs.

Query	Shredding	Ferry
T_1	13	25
T_2	42	5672
T_3	13	3654
T_4	16	7276

Even for a fairly simple nested query T_1, our approach performs almost twice as fast; for queries involving complex nesting structure with unions inside of nested comprehensions (T_2, T_3) or emptiness testing (T_4), the queries generated by Ferry can be up to 100–

400 times slower than our approach. The main performance problem with the Ferry approach as currently implemented appears to be that filtering often happens later than it should. This means that subqueries effectively compute a cartesian product when a join would suffice. Our approach does not suffer from this problem, at least on the test queries we considered.

Further work is needed to extend our approach (or determine how to combine the approaches) to optimise performance and handle grouping.

9. RELATED AND FUTURE WORK

We have already surveyed related work on query normalisation and shredding in the Introduction. Besides Cooper [7], several authors have recently considered higher-order query languages. Benedikt et al. [2] and Vu and Benedikt [27] study the complexity of containment, equivalence and evaluation for higher-order queries over flat relations. Higher-order features are also being added to XQuery 3.0 [21]. The Ferry system has been interfaced with Links by Ulrich [25], and supports higher-order functions via defunctionalisation [20]; this technique can easily be adapted to work with our approach. By connecting to Ferry, 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.

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 [23] 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 [11].

10. CONCLUSION

Combining efficient database access with functional programming abstractions is challenging in part because of the limitations of flat database queries. 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) have tried to realise this, getting the details right is tricky. Our contribution is an alternative algorithm for shredding that handles higher-order queries over bags. We give a concrete and modular implementation strategy for shredding that is proved correct, implemented in Links, and should also be straightforward to extend and to incorporate into other language-integrated query systems.

11. REFERENCES

[1] A. Beckmann. Exact bounds for lengths of reductions in typed λ-calculus. *Journal of Symbolic Logic*, 66, 2001.

[2] M. Benedikt, G. Puppis, and H. Vu. Positive higher-order queries. In *PODS*. ACM, 2010.

[3] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8, 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] A. J. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *PLDI*, 2010.

[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*, 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), 2002.

[10] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: Database-supported program execution. In *SIGMOD*, June 2009.

[11] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe LINQ compilation. *PVLDB*, 3(1), 2010.

[12] S. Lindley. Extensional rewriting with sums. In *TLCA*, 2007.

[13] S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI*, 2012.

[14] S. Lindley and I. Stark. Reducibility and $\top\top$-lifting for computation types. In *TLCA*, 2005.

[15] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.

[16] A. Ohori and K. Ueno. Making Standard ML a practical database programming language. In *ICFP*, 2011.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[18] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1), 1992.

[19] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[20] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4), 1998.

[21] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML query language. W3C Working Draft, December 2011. `http://www.w3.org/TR/xquery-30/`.

[22] H. J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11, April 1986.

[23] D. Suciu and V. Tannen. Efficient compilation of high-level data parallel algorithms. In *SPAA*. ACM, 1994.

[24] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2), June 1967.

[25] A. Ulrich. A Ferry-based query backend for the Links programming language. Master's thesis, University of Tübingen, 2011.

[26] 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.

[27] H. Vu and M. Benedikt. Complexity of higher-order queries. In *ICDT*. ACM, 2011.

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

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

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

[31] XML Query and XSL Working Groups. XQuery 1.0: An XML query language. Available at `http://www.w3.org/TR/xquery/`, 2007.

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:

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

and similarly

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

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 $O(|\textbf{adom}| * |R| + |\textbf{adom}|^2 * |S|)$ 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 $|\textbf{adom}| = 6$, 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 mested result, but doing so appears no easier than developing an alternative, direct algorithm.

B. TYPING RULES

The (standard) typing rules for λ_{NRC} queries are shown in Figure 8. The typing rules for shredded terms, packages, and values are shown in Figures 9, 10, 11 and 12.

C. QUERY NORMALISATION

Following our previous work [13], we separate query normalisation into a rewriting phase and a type-directed structurally recursive function. Where we diverge from our previous work is that we extend the rewrite relation to hoist all conditionals up to the nearest conditional (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, β-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 allow them to be converted to where clauses.
- The third and final stage hoists all unions up to the top-level, η-expands tables and variables, and turns all conditionals into where clauses.

Weak normalisation and strong normalisation. Given a term M and a rewrite relation $\rightsquigarrow$, we write $M \not\rightsquigarrow$ if M is irreducible, that is no rewrite rules in $\rightsquigarrow$ 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 $\rightsquigarrow_r$, or *r-WN*, if there exists a finite reduction sequence

$$M \rightsquigarrow M_1 \rightsquigarrow \ldots \rightsquigarrow M_n \not\rightsquigarrow$$

A term M is *strongly normalising* with respect to a rewrite relation $\rightsquigarrow_r$, or *r-SN*, if every reduction sequence starting from M is finite.

Var
$$\frac{}{\Gamma, x : A \vdash x : A}$$

Constant
$$\frac{\Sigma(c) = \langle O_1, \dots, O_n \rangle \to O' \qquad [\Gamma \vdash M_i : O_i]_{i=1}^n}{\Gamma \vdash c(M_1, \dots, M_n) : O'}$$

Lam
$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B}$$

App
$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

Record
$$\frac{[\Gamma \vdash M_i : A_i]_{i=1}^n}{\Gamma \vdash \langle \ell_i = M_i \rangle_{i=1}^n : \langle \ell_i = A_i \rangle_{i=1}^n}$$

Project
$$\frac{\Gamma \vdash M : \langle \ell_i : A_i \rangle_{i=1}^n}{\Gamma \vdash M.\ell_j : A_j}$$

If
$$\frac{\Gamma \vdash M : \mathit{Bool} \qquad \Gamma \vdash N : A \qquad \Gamma \vdash N' : A}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ N' : A}$$

Empty
$$\frac{}{\Gamma \vdash \emptyset : \mathrm{Bag}\ A}$$

Singleton
$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{return}\ M : \mathrm{Bag}\ A}$$

Union
$$\frac{\Gamma \vdash M : \mathrm{Bag}\ A \qquad \Gamma \vdash N : \mathrm{Bag}\ A}{\Gamma \vdash M \uplus N : \mathrm{Bag}\ A}$$

For
$$\frac{\Gamma \vdash M : \mathrm{Bag}\ A \qquad \Gamma, x : A \vdash N : \mathrm{Bag}\ B}{\Gamma \vdash \mathsf{for}\,(x \leftarrow M)\ N : \mathrm{Bag}\ B}$$

IsEmpty
$$\frac{\Gamma \vdash M : \mathrm{Bag}\ A}{\Gamma \vdash \mathsf{empty}\ M : \mathit{Bool}}$$

Table
$$\frac{\Sigma(t) = \mathrm{Bag}\ \langle \overrightarrow{\ell : O} \rangle}{\Gamma \vdash \mathsf{table}\ t : \mathrm{Bag}\ \langle \overrightarrow{\ell : O} \rangle}$$

Figure 8: Typing rules for higher-order nested queries

Var
$$\frac{}{\Gamma, x : \langle \overrightarrow{\ell : F} \rangle \vdash x : \langle \overrightarrow{\ell : F} \rangle}$$

Constant
$$\frac{\Sigma(c) = \langle O_1, \dots, O_n \rangle \to O' \qquad [\Gamma \vdash X_i : O_i]_{i=1}^n}{\Gamma \vdash c(X_1, \dots, X_n) : O'}$$

Record
$$\frac{[\Gamma \vdash N_i : A_i]_{i=1}^n}{\Gamma \vdash \langle \ell_i = N_i \rangle_{i=1}^n : \langle \ell_i = A_i \rangle_{i=1}^n}$$

Project
$$\frac{\Gamma \vdash x : \langle \ell_i : A_i \rangle_{i=1}^n}{\Gamma \vdash x.\ell_j : A_j}$$

Index
$$\frac{}{\Gamma \vdash a \diamond d : \mathit{Index}}$$

Singleton
$$\frac{\Gamma \vdash I : \mathit{Index} \qquad \Gamma \vdash N : F}{\Gamma \vdash \mathsf{return}^a\ \langle I, N \rangle : \mathrm{Bag}\ \langle \mathit{Index}, F \rangle}$$

Union
$$\frac{[\Gamma \vdash C_i : \mathrm{Bag}\ \langle \mathit{Index}, F \rangle]_{i=1}^n}{\Gamma \vdash \biguplus \vec{C} : \mathrm{Bag}\ \langle \mathit{Index}, F \rangle}$$

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

IsEmpty
$$\frac{\Gamma \vdash L : \mathrm{Bag}\ \langle \mathit{Index}, F \rangle}{\Gamma \vdash \mathsf{empty}\ L : \mathit{Bool}}$$

Figure 9: Typing rules for shredded terms

Base
$$\frac{}{\vdash O(\Lambda_S) : O(\mathcal{T}_S)}$$

Record
$$\frac{[\vdash \hat{A}_i(\Lambda_S) : \hat{A}'_i(\mathcal{T}_S)]_{i=1}^n}{\vdash \langle \ell_i : \hat{A}_i(\Lambda_S) \rangle_{i=1}^n : \langle \ell_i : \hat{A}'_i(\mathcal{T}_S) \rangle_{i=1}^n}$$

Bag
$$\frac{\vdash \hat{A}(\Lambda_S) : \hat{A}'(\mathcal{T}_S) \qquad \vdash L : A \qquad \mathit{erase}(\hat{A}(\Lambda_S)) = A}{\vdash (\mathrm{Bag}\ \hat{A}(\Lambda_S))^L : (\mathrm{Bag}\ \hat{A}'(\mathcal{T}_S))^A}$$

Figure 10: Typing rules for shredded packages

Result
$$\frac{[\vdash v_i : A_i]_{i=1}^n \qquad [\vdash J : \mathit{Index}]_{i=1}^n}{\vdash [v_i @ J_i]_{i=1}^n : \mathrm{Bag}\ A}$$

Record
$$\frac{[\vdash v_i : A_i]_{i=1}^n}{\vdash \langle \ell_i = v_i \rangle_{i=1}^n : \langle \ell_i = A_i \rangle_{i=1}^n}$$

Constant
$$\frac{\Sigma(c) = O}{\vdash c : O}$$

Figure 11: Typing rules for indexed nested values

Result
$$\frac{[\vdash I_i : \mathit{Index}]_{i=1}^n \qquad [\vdash w_i : F]_{i=1}^n \qquad [\vdash J_i : \mathit{Index}]_{i=1}^n}{\vdash \langle I_1, w_1 @ J_1 \rangle, \dots, \langle I_n, w_n @ J_n \rangle : \mathrm{Bag}\ \langle \mathit{Index}, F \rangle}$$

Constant
$$\frac{\Sigma(c) = O}{\vdash c : O}$$

Record
$$\frac{[\vdash n_i : F_i]_{i=1}^n}{\vdash \langle \ell_i = w_i \rangle_{i=1}^n : \langle \ell_i = F_i \rangle_{i=1}^n}$$

Figure 12: Typing rules for shredded values

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 $\leadsto_r$.

We now describe each normalisation stage in turn.

C.1 Symbolic evaluation

The β-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.

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

Fundamentally, β-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 β-rule above), the introduction form is a lambda and the elimination form is an application. Applying a β-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 β-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 β-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 β-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 [19, 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 λ_{NRC} are as follows.

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

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 β-reductions. For instance,

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

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

$$\begin{aligned}E[\textsf{for}\,(x \leftarrow M)\; N] &\rightsquigarrow_c \textsf{for}\,(x \leftarrow M)\; E[N] \\ E[\textsf{if } L \textsf{ then } M \textsf{ else } N] &\rightsquigarrow_c \textsf{if } L \textsf{ then } E[M] \textsf{ else } E[N] \\ E[\emptyset] &\rightsquigarrow_c \emptyset \\ E[M_1 \uplus M_2] &\rightsquigarrow_c E[M_1] \uplus E[M_2]\end{aligned}$$

For example:

$$(\textsf{if } L \textsf{ then } M_1 \textsf{ else } M_2)\; M \rightsquigarrow_c \textsf{if } L \textsf{ then } M_1\; M \textsf{ else } M_2\; M$$

Note that some combinations of elimination frames and rewrite rule are impossible in a well-typed term, such as $\textsf{if } \emptyset \textsf{ then } M \textsf{ else } N$. For the purposes of reduction we treat $\textsf{empty}$ 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 our previous proof of strong normalisation for simply-typed λ-calculus with sums [12], which generalises the $\top\top$-lifting approach [14], which in turn extends Tait's proof of strong normalisation for simply-typed λ- calculus [24].

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 our previous work [12] 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.

$$S \rightsquigarrow_c S' \quad \overset{def}{\Longleftrightarrow} \quad \forall M.S[M] \rightsquigarrow_c S'[M] \quad \Longleftrightarrow \quad S[x] \rightsquigarrow_c S'[x]$$

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

PROOF. Induction on the structure of S. □

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.
- $S \circ ([\;].\ell) : (A \times B) \multimap C$ is reducible if S is reducible.
- $S : \text{Bag}\, A \multimap C$ is reducible if $S[\textsf{return}\; M]$ is c-SN for all reducible $M : A$.
- $S : Bool \multimap C$ is reducible if $S[\textsf{true}]$ is c-SN and $S[\textsf{false}]$ is c-SN.
- $M : A$ is reducible if $S[M]$ is c-SN for all reducible $S : A \multimap C$.

LEMMA 12. *If $M : A$ is reducible then M is c-SN.*

PROOF. Follows immediately from reducibility of Id and the definition of reducibility on terms. □

LEMMA 13. *$x : A$ is reducible.*

PROOF. By induction on A using Lemma 11 and Lemma 12. □

COROLLARY 14. *If $S : A \multimap C$ is reducible then S is c-SN.*

Each type constructor has an associated β-rule. Each β-rule gives rise to an SN-closure property.

LEMMA 15 (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 $S[\textsf{for}\,(x \leftarrow \textsf{return}\; M)\; N]$ is c-SN.*

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

□

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

LEMMA 16 (REDUCIBILITY-CLOSURE).

($\rightarrow$) *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* for $(x \leftarrow M)\, N$ *is reducible.*

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

PROOF. Each property follows from the corresponding part of Lemma 15 using Lemma 12 and Corollary 14. □

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

LEMMA 17 (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.

□

THEOREM 18. *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 16 and Lemma 17. □

THEOREM 19 (STRONG NORMALISATION). *The relation $\rightsquigarrow_c$ is strongly normalising.*

PROOF. Let M be a term with free variables $\vec{x}$. By Lemma 13, $\vec{x}$ are reducible. Hence, by Theorem 18, M is c-SN. □

It is well known that β-reduction in simply-typed λ-calculus has non-elementary complexity in the worst case [1]. The relation $\rightsquigarrow_c$ includes β-reduction, so it must be at least as bad (we conjecture that it has the same asymptotic complexity, as $\rightsquigarrow_c$ can be reduced to β-reduction on simply-typed λ-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:

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

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:

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

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

LEMMA 20.

1. *If M, N, N' are h-SN then* if M then N 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. □

PROPOSITION 21. *The relation $\rightsquigarrow_h$ is strongly normalising.*

PROOF. By induction on the structure of terms using Lemma 20. □

C.3 The query normalisation function

The following definition of the function $norm$ generalises the normalisation algorithm from our previous work [13].

$$norm_A(M) = (\!|N|\!)_A$$

where $M \rightsquigarrow_c^* M' \rightsquigarrow_h^* N$ with $M' \not\rightsquigarrow_c$, $N \not\rightsquigarrow_h$ and:

$$\begin{aligned}
(\!|c([X_i : O_1]_{i=1}^n)|\!)_O &= c([(\!|X_i|\!)_{O_i}]_{i=1}^n) \\
(\!|x.\ell|\!)_O &= x.\ell \\
(\!|\mathsf{empty}\ M : \mathrm{Bag}\ A|\!)_{Bool} &= \mathsf{empty}\ (\!|M|\!)_{\mathrm{Bag}\ A} \\
(\!|M|\!)_{\langle \ell_i : A_i\rangle_{i=1}^n} &= \langle \ell_i = \mathcal{F}(\!|M|\!)_{A_i,\ell_i}\rangle_{i=1}^n \\
(\!|M|\!)_{\mathrm{Bag}\ A} &= \biguplus (\mathcal{B}(\!|M|\!)^\star_{A,[\,],\mathsf{true}})
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{F}(\!|x|\!)_{A,\ell_i} &= (\!|x.\ell_i|\!)_A \\
\mathcal{F}(\!|\langle \ell_i = M_i\rangle_{i=1}^n|\!)_{A,\ell_i} &= (\!|M_i|\!)_A
\end{aligned}$$

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 22. *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 $(\!|-|\!)$, $\mathcal{B}(\!|-|\!)^\star$, and $\mathcal{F}(\!|-|\!)$ are structurally recursive (modulo expanding out the right-hand-side of the definition of $\mathcal{B}(\!|\mathsf{table}\ t|\!)^\star_{A,\vec{G},L}$). □

D. PROOF OF CORRECTNESS OF SHREDDING

We begin with a simple lemma that states that the inner shredding function $\llbracket - \rrbracket$ commutes with the semantics.

LEMMA 23. $\mathcal{S}\llbracket\llbracket M\rrbracket_a\rrbracket_{\rho,\iota} = \llbracket\mathcal{A}\llbracket M\rrbracket_{\rho,\iota}\rrbracket_{a\diamond\iota}$

PROOF. By induction on the structure of M.

Case $x.\ell$:

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

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

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

Case $\mathsf{empty}\ M$:

$$\begin{aligned}
&\mathcal{S}\llbracket\llbracket\mathsf{empty}\ M\rrbracket_a\rrbracket_{\rho,\iota}\\
=\ &\mathcal{S}\llbracket\mathsf{empty}\ \llbracket M\rrbracket_\epsilon\rrbracket_{\rho,\iota}\\
=\ &\mathcal{A}\llbracket\mathsf{empty}\ \llbracket M\rrbracket_\epsilon\rrbracket_{\rho,\iota}\\
=\ &\llbracket\mathcal{A}\llbracket\mathsf{empty}\ M\rrbracket_{\rho,\iota}\rrbracket_{a\diamond\iota}
\end{aligned}$$

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

$$\begin{aligned}
&\mathcal{S}\llbracket\llbracket\langle\ell_i = M_i\rangle_{i=1}^n\rrbracket_a\rrbracket_{\rho,\iota}\\
=\ &\mathcal{S}\llbracket\langle\ell_i = \llbracket M_i\rrbracket_a\rangle_{i=1}^n\rrbracket_{\rho,\iota}\\
=\ &\langle\ell_i = \mathcal{S}\llbracket\llbracket M_i\rrbracket_a\rrbracket_{\rho,\iota}\rangle_{i=1}^n\\
=\ &\quad\text{(Induction hypothesis)}\\
&\langle\ell_i = \llbracket\mathcal{A}\llbracket M_i\rrbracket_{\rho,\iota}\rrbracket_{a\diamond\iota}\rangle_{i=1}^n\\
=\ &\llbracket\langle\ell_i = \mathcal{A}\llbracket M_i\rrbracket_{\rho,\iota}\rangle_{i=1}^n\rrbracket_{a\diamond\iota}\\
=\ &\llbracket\mathcal{A}\llbracket\langle\ell_i = M_i\rangle_{i=1}^n\rrbracket_{\rho,\iota}\rrbracket_{a\diamond\iota}
\end{aligned}$$

Case L:

$$\begin{aligned}
&\mathcal{S}\llbracket\llbracket L\rrbracket_a\rrbracket_{\rho,\iota}\\
=\ &\mathcal{S}\llbracket a\diamond\mathsf{in}\rrbracket_{\rho,\iota}\\
=\ &a\diamond\iota\\
=\ &\llbracket\mathcal{A}\llbracket L\rrbracket_{\rho,\iota}\rrbracket_{a\diamond\iota}
\end{aligned}$$

□

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 24.
1. $\mathcal{S}\llbracket\biguplus\llbracket\biguplus_{i=1}^n C_i\rrbracket^\star_{a,p}\rrbracket_{\rho,\iota} = concat([\mathcal{S}\llbracket\llbracket C_i\rrbracket^\star_{a,p}\rrbracket_{\rho,\iota}]_{i=1}^n)$
2. $\llbracket\mathcal{A}\llbracket\biguplus_{i=1}^n C_i\rrbracket_{\rho,\iota}\rrbracket^\star_{a\diamond\iota,p} = concat([\llbracket\mathcal{A}\llbracket C_i\rrbracket_{\rho,\iota}\rrbracket^\star_{a\diamond\iota,p}]_{i=1}^n)$

PROOF.

1.

$$\begin{aligned}
&\mathcal{S}\llbracket\biguplus\llbracket\biguplus_{i=1}^n C_i\rrbracket^\star_{a,p}\rrbracket_{\rho,\iota}\\
=\ &\mathcal{S}\llbracket\biguplus\ concat([\llbracket C_i\rrbracket^\star_{a,p}]_{i=1}^n)\rrbracket_{\rho,\iota}\\
=\ &\mathcal{S}\llbracket\biguplus\ concat([\llbracket C\rrbracket^\star_{a,p} \mid C \leftarrow \vec{C}])\rrbracket_{\rho,\iota}\\
=\ &concat(concat([\mathcal{S}\llbracket C'\rrbracket_{\rho,\iota} \mid C \leftarrow \vec{C}, C' \leftarrow \llbracket C\rrbracket^\star_{a,p}]))\\
=\ &concat([\mathcal{S}\llbracket\biguplus\llbracket C\rrbracket^\star_{a,p}\rrbracket_{\rho,\iota} \mid C \leftarrow \vec{C}])\\
=\ &concat([\mathcal{S}\llbracket\biguplus\llbracket C_i\rrbracket^\star_{a,p}\rrbracket_{\rho,\iota}]_{i=1}^n)
\end{aligned}$$

2.

$$\begin{aligned}
&\llbracket\mathcal{A}\llbracket\biguplus_{i=1}^n C_i\rrbracket_{\rho,\iota}\rrbracket^\star_{a\diamond\iota,p}\\
=\ &\llbracket concat([\mathcal{A}\llbracket C_i\rrbracket_{\rho,\iota}]_{i=1}^n)\rrbracket^\star_{a\diamond\iota,p}\\
=\ &\llbracket concat([\mathcal{A}\llbracket C\rrbracket_{\rho,\iota} \mid C \leftarrow \vec{C}])\rrbracket^\star_{a\diamond\iota,p}\\
=\ &concat(concat([\llbracket s\rrbracket^\star_{a\diamond\iota,p} \mid C \leftarrow \vec{C}, s \leftarrow \mathcal{A}\llbracket C\rrbracket_{\rho,\iota}]))\\
=\ &concat([\llbracket\mathcal{A}\llbracket C\rrbracket_{\rho,\iota}\rrbracket^\star_{a\diamond\iota,p} \mid C \leftarrow \vec{C}])\\
=\ &concat([\llbracket\mathcal{A}\llbracket C_i\rrbracket_{\rho,\iota}\rrbracket^\star_{a\diamond\iota,p}]_{i=1}^n)
\end{aligned}$$

□

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

LEMMA 25. $\mathcal{S}\llbracket \lceil\!\lceil L \rceil\!\rceil_p \rrbracket_{\rho,1} = \lceil\!\lceil \mathcal{A}\llbracket L \rrbracket_{\rho,1} \rceil\!\rceil_p$

PROOF. We prove the following:

1. $\mathcal{S}\llbracket \lceil\!\lceil L \rceil\!\rceil_p \rrbracket_{\rho,1} = \lceil\!\lceil \mathcal{A}\llbracket L \rrbracket_{\rho,1} \rceil\!\rceil_p$
2. $\mathcal{S}\llbracket \biguplus \lceil\!\lceil C \rceil\!\rceil^\star_{a,p} \rrbracket_{\rho,\iota} = \lceil\!\lceil \mathcal{A}\llbracket C \rrbracket_{\rho,\iota} \rceil\!\rceil^\star_{a\diamond\iota,p}$
3. $\mathcal{S}\llbracket \lceil\!\lceil M \rceil\!\rceil^\star_{a,p} \rrbracket_{\rho,\iota} = \lceil\!\lceil \mathcal{A}\llbracket M \rrbracket_{\rho,\iota} \rceil\!\rceil^\star_{a\diamond\iota,p}$

The first equation is the result we require. Observe that it follows from (2) and Lemma 24. 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 ϵ:

$$
\begin{aligned}
& \mathcal{S}\llbracket \biguplus \lceil\!\lceil \mathsf{for}\ (\overrightarrow{x \leftarrow t}\ \mathsf{where}\ X)\ \mathsf{return}^b\ M \rceil\!\rceil^\star_{a,\epsilon} \rrbracket_{\rho,\iota} \\
= & \quad \text{(Definition of } \mathcal{S}\llbracket - \rrbracket) \\
& \mathcal{S}\llbracket \mathsf{for}\ (\overrightarrow{x \leftarrow t}\ \mathsf{where}\ X)\ \mathsf{return}^b\ \langle a \diamond \iota, \lfloor\!\lfloor M \rfloor\!\rfloor_b \rangle \rrbracket_{\rho,\iota} \\
= & \quad \text{(Definition of } \mathcal{S}\llbracket - \rrbracket) \\
& [\langle a \diamond \iota, \mathcal{S}\llbracket \lfloor\!\lfloor M \rfloor\!\rfloor_b \rrbracket_{\rho[\overrightarrow{x \mapsto v}],\iota.i} \rangle @ b \diamond \iota.i \\
& \quad \mid \langle i, \vec{v} \rangle \leftarrow enum([\vec{v} \mid \overrightarrow{v \leftarrow \llbracket t \rrbracket}, \mathcal{S}\llbracket X \rrbracket_{\rho[\overrightarrow{x \mapsto v}]}])] \\
= & \quad \text{(Lemma 23)} \\
& [\langle a \diamond \iota, \lfloor\!\lfloor \mathcal{A}\llbracket M \rrbracket_{\rho[\overrightarrow{x \mapsto v}],\iota.i} \rfloor\!\rfloor_{b\diamond\iota.i} \rangle @ b \diamond \iota.i \\
& \quad \mid \langle i, \vec{v} \rangle \leftarrow enum([\vec{v} \mid \overrightarrow{v \leftarrow \llbracket t \rrbracket}, \mathcal{S}\llbracket X \rrbracket_{\rho[\overrightarrow{x \mapsto v}]}])] \\
= & \quad \text{(Definition of } \lceil\!\lceil - \rceil\!\rceil^\star) \\
& \lceil\!\lceil [\mathcal{A}\llbracket M \rrbracket_{\rho[\overrightarrow{x \mapsto v}],\iota.i} @ b \diamond \iota.i \\
& \quad \mid \langle i, \vec{v} \rangle \leftarrow enum([\vec{v} \mid \overrightarrow{v \leftarrow \llbracket t \rrbracket}, \mathcal{S}\llbracket X \rrbracket_{\rho[\overrightarrow{x \mapsto v}]}])] \rceil\!\rceil^\star_{a\diamond\iota,\epsilon} \\
= & \quad \text{(Definition of } \mathcal{A}\llbracket - \rrbracket) \\
& \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,\epsilon}
\end{aligned}
$$

Case $\downarrow.\epsilon$:

$$
\begin{aligned}
& \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) \\
& concat([concat \\
& \qquad\qquad ([\mathcal{S}\llbracket C \rrbracket_{\rho[\overrightarrow{x \mapsto v}],\iota.i} \\
& \qquad\qquad\quad \mid \langle i, \vec{v} \rangle \leftarrow enum([\vec{v} \mid \overrightarrow{v \leftarrow \llbracket t \rrbracket}, \mathcal{S}\llbracket X \rrbracket_{\rho[\overrightarrow{x \mapsto v}]}])]) \\
& \qquad\quad \mid C \leftarrow \lceil\!\lceil M \rceil\!\rceil^\star_{b,p}]) \\
= & \quad \text{(Definition of } \mathcal{S}\llbracket - \rrbracket) \\
& 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 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))} \\
& 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 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{aligned}
$$

There are three cases for (3).

Cases ϵ and $\downarrow.\epsilon$: follow from (2) by applying the two parts of Lemma 24 to the left and right-hand side respectively.

Case $\ell_i.\epsilon$:

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

□

We now lift Lemma 25 to shredded packages.

PROOF OF THEOREM 4. We need to show that if $\vdash L : \mathrm{Bag}\ A$ then

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

This is straightforward by induction on A, using Lemma 25. □

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.

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

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

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

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

Case O:

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

$$
\begin{aligned}
& [(\mathbin{/\!\!/} s \mathbin{\backslash\!\!\backslash}_{J',\downarrow.\vec{\ell},c} @I_{\mathsf{in}} \\
&\qquad \mid \langle I_{\mathsf{out}}, c\rangle @I_{\mathsf{in}} \leftarrow \llbracket s \rrbracket^{\star}_{J',\vec{\ell}}, I_{\mathsf{out}} = I] \\
= \quad & \text{(Definition of } \mathbin{/\!\!/}-\mathbin{\backslash\!\!\backslash}) \\
& [c @I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, c\rangle @I_{\mathsf{in}} \leftarrow \llbracket s \rrbracket^{\star}_{J',\vec{\ell}}, I_{\mathsf{out}} = I] \\
= \quad & (J' = I) \\
& s \\
= \quad & \text{(Definition of } \mathbin{/\!\!/}-\mathbin{\backslash\!\!\backslash}) \\
& [v'.\vec{\ell} @I_{\mathsf{in}} \mid v' @I_{\mathsf{in}} \leftarrow \mathbin{/\!\!/} s \mathbin{\backslash\!\!\backslash}_{J',\epsilon,I}]
\end{aligned}
$$

Subcase $\ell_i.p$:

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

Subcase $\downarrow.p$:

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

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 $\mathit{zip}_{\vec{\ell}}$, for transforming a record of lists of equal length to a list of records, and $\mathit{unzip}_{\vec{\ell}}$, 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.

$$
\begin{aligned}
\mathit{zip}_{\vec{\ell}}\langle \ell_i = [\,]\rangle_{i=1}^{n} &= [\,] \\
\mathit{zip}_{\vec{\ell}}\langle \ell_i = v_i @J :: s_i\rangle_{i=1}^{n} &= \langle \ell_i = v_i\rangle_{i=1}^{n} @J :: \mathit{zip}_{\vec{\ell}}\langle \ell_i = s_i\rangle_{i=1}^{n} \\
\mathit{unzip}_{\vec{\ell}}(s) &= \langle \ell_i = [v.\ell_i @J \mid v @J \leftarrow s]\rangle_{i=1}^{n}
\end{aligned}
$$

If $\vec{\ell}$ is a non-empty list of column labels then $\mathit{zip}_{\vec{\ell}}$ is the inverse of $\mathit{unzip}_{\vec{\ell}}$.

$$
\mathit{zip}_{\vec{\ell}}(\mathit{unzip}_{\vec{\ell}}(s)) = s, \qquad \text{if } |\vec{\ell}| \geq 1
$$

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

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

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

Subcase $\ell_i.p$:

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

Subcase $\downarrow.p$:

$[\mathbin{/\!\!/}[v_i@J_i]_{i=1}^n\mathbin{\backslash\!\!\backslash}_{J',\downarrow.p.\downarrow.\vec{\ell},I_{\mathsf{in}}}@I_{\mathsf{in}}$
$\quad \mid \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow \llbracket[v_i@J_i]_{i=1}^n\rrbracket^\star_{I_{\mathsf{out}},\downarrow.p.\vec{\ell}}, I_{\mathsf{out}} = I]$

$=$ (Definitions of $\mathbin{/\!\!/}-\mathbin{\backslash\!\!\backslash}$ and $\llbracket-\rrbracket^\star$)

$[(concat([\mathbin{/\!\!/}v_i\mathbin{\backslash\!\!\backslash}_{J_i,p.\downarrow.\vec{\ell},I_{\mathsf{in}}}]))@I_{\mathsf{in}}$
$\quad \mid \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow concat([\llbracket v_i\rrbracket^\star_{J_i,p.\vec{\ell}}]_{i=1}^n), I_{\mathsf{out}} = I]$

$=$ (Expanding $concat([\llbracket-\rrbracket^\star]_{i=1}^n)$)

$[(concat([\mathbin{/\!\!/}v_i\mathbin{\backslash\!\!\backslash}_{J_i,p.\downarrow.\vec{\ell},I_{\mathsf{in}}}]))@I_{\mathsf{in}}$
$\quad \mid v_i@J_i \leftarrow [v_i@J_i]_{i=1}^n,$
$\quad\quad \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow \llbracket v_i\rrbracket^\star_{J_i,p.\vec{\ell}}, I_{\mathsf{out}} = I]$

$=$ (v is well-indexed)

$[\mathbin{/\!\!/}v_i\mathbin{\backslash\!\!\backslash}_{J_i,p.\downarrow.\vec{\ell},I_{\mathsf{in}}}@I_{\mathsf{in}}$
$\quad \mid v_i@J_i \leftarrow [v_i@J_i]_{i=1}^n,$
$\quad\quad \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow \llbracket v_i\rrbracket^\star_{J_i,p.\vec{\ell}}, I_{\mathsf{out}} = I]$

$=$ (Comprehension $\longrightarrow$ concatenation)

$concat$
$\quad([\mathbin{/\!\!/}v_i\mathbin{\backslash\!\!\backslash}_{J_i,p.\downarrow.\vec{\ell},I_{\mathsf{in}}}@I_{\mathsf{in}}$
$\quad\quad \mid \langle I_{\mathsf{out}}, I_{\mathsf{in}}\rangle@I_{\mathsf{in}} \leftarrow \llbracket v_i\rrbracket^\star_{J_i,p.\vec{\ell}}, I_{\mathsf{out}} = I]_{i=1}^n)$

$=$ (Induction hypothesis)

$concat([v'.\vec{\ell}@I_{\mathsf{in}} \mid v'@I_{\mathsf{in}} \leftarrow \mathbin{/\!\!/}v_i\mathbin{\backslash\!\!\backslash}_{J_i,\downarrow.p,I}]_{i=1}^n)$

$=$ (Concatenation $\longrightarrow$ comprehension)

$[v'.\vec{\ell}@I_{\mathsf{in}} \mid v'@I_{\mathsf{in}} \leftarrow \mathbin{/\!\!/}[v_i@J_i]_{i=1}^n\mathbin{\backslash\!\!\backslash}_{J',\downarrow.p,I}]$

□

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

PROOF. By induction on the structure of A.

Case O:

$$\begin{aligned} & stitch_c(shred_{s,p}(O)) \\ = \; & c \\ = \; & \mathbin{/\!\!/}s\mathbin{\backslash\!\!\backslash}_{p,c} \end{aligned}$$

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

$$\begin{aligned} & stitch_{\langle \ell_i = w_i\rangle_{i=1}^n}(shred_{s,p}(\langle \ell_i : A\rangle_{i=1}^n)) \\ = \; & \langle \ell_i = stitch_{w_i}(shred_{s,p.l}(A))\rangle_{i=1}^n \\ = \; & \text{(Induction hypothesis)} \\ & \langle \ell_i = \mathbin{/\!\!/}s\mathbin{\backslash\!\!\backslash}_{p.l,w_i}\rangle_{i=1}^n \\ = \; & \mathbin{/\!\!/}s\mathbin{\backslash\!\!\backslash}_{p,\langle \ell_i = w_i\rangle_{i=1}^n} \end{aligned}$$

Case Bag A:

$$\begin{aligned} & stitch_I(shred_{s,p}(\mathrm{Bag}\ A)) \\ = \; & stitch_I(((shred_{s,p.\downarrow}(A))^{\llbracket s\rrbracket_p})) \\ = \; & [(stitch_n(shred_{s,p.\downarrow}(A))@I_{\mathsf{in}}) \\ & \quad \mid \langle I_{\mathsf{out}}, w\rangle@I_{\mathsf{in}} \leftarrow \llbracket s\rrbracket^\star_p, I_{\mathsf{out}} = I] \\ = \; & \text{(Induction hypothesis)} \\ & [\mathbin{/\!\!/}s\mathbin{\backslash\!\!\backslash}_{p.\downarrow,w}@I_{\mathsf{in}} \mid \langle I_{\mathsf{out}}, w\rangle@I_{\mathsf{in}} \leftarrow \llbracket s\rrbracket^\star_p, I_{\mathsf{out}} = I] \\ = \; & \text{(Lemma 26)} \\ & \mathbin{/\!\!/}s\mathbin{\backslash\!\!\backslash}_{p,I} \end{aligned}$$

□

PROOF OF THEOREM 6. By Theorem 27, setting $p = \epsilon$ and $n = \top \diamond 1$. □

We now obtain the main result.

PROOF OF THEOREM 7. Recall the statement of Theorem 7: we need to show that $\vdash L : \mathrm{Bag}\ A$ then:

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

and in particular:

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

This follows immediately from Theorem 4, Lemma 5 and Theorem 6. □

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.

Types	$A, B ::= \mathrm{Bag}\ \langle\overrightarrow{\ell : O}\rangle$
Base types	$O ::= Int \mid Bool \mid String \mid \langle\rangle$

Our 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.

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

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

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

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 • 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.

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

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

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

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

Type soundness.

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

Flat values.

$$
\begin{array}{lrl}
\text{Results} & s & ::= \vec{r} \\
\text{Rows} & r & ::= \langle \ell_1 = c_1, \ldots, \ell_n = c_n \rangle
\end{array}
$$

Semantics of flat record queries.

$$\mathcal{R}[\![L]\!] = \mathcal{L}[\![L]\!]$$

Unflattening record values.

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

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

Type soundness.

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

Correctness

PROPOSITION 28. *If L is a let-inserted query and* $\vdash L : \mathrm{Bag}\ A$, *then*

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

F. SAMPLE QUERIES

We tested the shredded queries generated by our shredding algorithm and by Ferry on four queries T_1, T_2, T_3, T_4. The queries T_1, T_2, and T_4 are over the following schema:

$$
\begin{aligned}
\Sigma(t_1) &= \mathrm{Bag}\ \langle \mathsf{v} : \mathit{Int}, \mathsf{a} : \mathit{Int} \rangle \\
\Sigma(t_2) &= \mathrm{Bag}\ \langle \mathsf{a} : \mathit{Int}, \mathsf{b} : \mathit{Int} \rangle \\
\Sigma(t_3) &= \mathrm{Bag}\ \langle \mathsf{b} : \mathit{Int}, \mathsf{v} : \mathit{Int} \rangle \\
\Sigma(t_4) &= \mathrm{Bag}\ \langle \mathsf{b} : \mathit{Int}, \mathsf{v} : \mathit{Int} \rangle
\end{aligned}
$$

The query T_3 is over the following schema:

$$
\begin{aligned}
\Sigma(t_1) &= \mathrm{Bag}\ \langle \mathsf{v} : \mathit{Int}, a_1 : \mathit{Int}, a_2 : \mathit{Int} \rangle \\
\Sigma(t_2) &= \mathrm{Bag}\ \langle \mathsf{a} : \mathit{Int}, \mathsf{v} : \mathit{Int}, b_1 : \mathit{Int}, b_2 : \mathit{Int} \rangle \\
\Sigma(t_3) &= \mathrm{Bag}\ \langle \mathsf{b} : \mathit{Int}, \mathsf{v} : \mathit{Int} \rangle \\
\Sigma(t_4) &= \mathrm{Bag}\ \langle \mathsf{b} : \mathit{Int}, \mathsf{v} : \mathit{Int} \rangle
\end{aligned}
$$

The queries are as follows:

$$
\begin{array}{l}
T_1 : \mathrm{Bag}\,(\mathrm{Bag}\,(\mathrm{Bag}\,\mathit{Int})) \\
T_1 = \mathsf{for}\,(x \leftarrow t_1) \\
\qquad \mathsf{return}\,(\mathsf{for}\,(y \leftarrow t_2)\ \mathsf{where}\,(x.\mathsf{a} = y.\mathsf{a}) \\
\qquad\quad \mathsf{return}\,(\mathsf{for}\,(z \leftarrow t_3) \\
\qquad\qquad\qquad \mathsf{where}\,(x.\mathsf{b} = y.\mathsf{b})\ \mathsf{return}\ z.\mathsf{v}))
\end{array}
$$

$$
\begin{array}{l}
T_2 : \mathrm{Bag}\,(\mathrm{Bag}\,(\mathrm{Bag}\,\mathit{Int})) \\
T_2 = \mathsf{for}\,(x \leftarrow t_1) \\
\qquad \mathsf{return}\,(\mathsf{for}\,(y \leftarrow t_2)\ \mathsf{where}\,(x.\mathsf{a} = y.\mathsf{a}) \\
\qquad\qquad \mathsf{return}\,(\mathsf{for}\,(z \leftarrow t_3)\ \mathsf{where}\,(x.\mathsf{b} = y.\mathsf{b}) \\
\qquad\qquad\qquad \mathsf{return}\ z.\mathsf{v}) \\
\qquad\quad \uplus \\
\qquad\quad \mathsf{for}\,(y \leftarrow t_2)\ \mathsf{where}\,(x.\mathsf{a} = y.\mathsf{a}) \\
\qquad\qquad \mathsf{return}\,(\mathsf{for}\,(z \leftarrow t_4)\ \mathsf{where}\,(x.\mathsf{b} = y.\mathsf{b}) \\
\qquad\qquad\qquad \mathsf{return}\ z.\mathsf{v})
\end{array}
$$

$$
\begin{array}{l}
T_3 : \mathrm{Bag}\ \langle \mathsf{v} : \mathit{Int}, a_1 : \mathrm{Bag}\ \langle \mathsf{v} : \mathit{Int}, \mathsf{a} : \mathrm{Bag}\ \mathit{Int} \rangle, \\
\qquad\qquad a_2 : \mathrm{Bag}\ \langle \mathsf{v} : \mathit{Int}, \mathsf{a} : \mathrm{Bag}\ \mathit{Int} \rangle \rangle \\
T_3 = \mathsf{for}\,(x \leftarrow t_1) \\
\qquad \mathsf{return}\,(\langle \mathsf{v} = x.\mathsf{v}, \\
\qquad\quad \mathsf{a} = \mathsf{for}\,(y \leftarrow t_2)\ \mathsf{where}\,(x.a_1 = y.\mathsf{a}) \\
\qquad\qquad \mathsf{return}\,(\langle \mathsf{v} = y.\mathsf{v}, \\
\qquad\qquad\quad \mathsf{b} = (\mathsf{for}\,(z \leftarrow t_3) \\
\qquad\qquad\qquad \mathsf{where}\,(y.b_1 = z.\mathsf{b}) \\
\qquad\qquad\qquad\quad \mathsf{return}\ z.\mathsf{v}) \\
\qquad\qquad\quad \uplus \\
\qquad\qquad\quad (\mathsf{for}\,(z \leftarrow t_4) \\
\qquad\qquad\qquad \mathsf{where}\,(y.b_2 = z.\mathsf{b}) \\
\qquad\qquad\qquad\quad \mathsf{return}\ z.\mathsf{v})\rangle)\rangle) \\
\quad \uplus \\
\quad \mathsf{for}\,(x \leftarrow t_1) \\
\qquad \mathsf{return}\,(\langle \mathsf{v} = x.\mathsf{v}, \\
\qquad\quad \mathsf{a} = \mathsf{for}\,(y \leftarrow t_2)\ \mathsf{where}\,(x.a_2 = y.\mathsf{a}) \\
\qquad\qquad \mathsf{return}\,(\langle \mathsf{v} = y.\mathsf{v}, \\
\qquad\qquad\quad \mathsf{b} = (\mathsf{for}\,(z \leftarrow t_3) \\
\qquad\qquad\qquad \mathsf{where}\,(y.b_1 = z.\mathsf{b}) \\
\qquad\qquad\qquad\quad \mathsf{return}\ z.\mathsf{v}) \\
\qquad\qquad\quad \uplus \\
\qquad\qquad\quad (\mathsf{for}\,(z \leftarrow t_4) \\
\qquad\qquad\qquad \mathsf{where}\,(y.b_2 = z.\mathsf{b}) \\
\qquad\qquad\qquad\quad \mathsf{return}\ z.\mathsf{v})\rangle)\rangle)
\end{array}
$$

$$
\begin{array}{l}
T_4 : \mathrm{Bag}\,(\mathrm{Bag}\,(\mathrm{Bag}\,\mathit{Int})) \\
T_4 = \mathsf{for}\,(x \leftarrow t_1) \\
\qquad \mathsf{return}\,(\mathsf{for}\,(y \leftarrow t_2)\ \mathsf{where}\,(x.\mathsf{a} = y.\mathsf{a}) \\
\qquad\qquad \mathsf{return}\,(\mathsf{if}\,(\mathsf{empty}\,(\mathsf{for}\,(z \leftarrow t_3) \\
\qquad\qquad\qquad\qquad \mathsf{where}\,(y.\mathsf{b} = z.\mathsf{b}) \\
\qquad\qquad\qquad\qquad\quad \mathsf{return}\ z.\mathsf{v})) \\
\qquad\qquad\qquad \{\!\{1, 2, 3, 4, 5\}\!\} \\
\qquad\qquad\qquad (\mathsf{for}\,(z \leftarrow t_3) \\
\qquad\qquad\qquad\quad \mathsf{where}\,(y.\mathsf{b} = z.\mathsf{b})\ \mathsf{return}\ z.\mathsf{v})))
\end{array}
$$

For T_1, we generated a regular three-dimensional array structure with dimensions $10 \times 100 \times 100$ populated by random positive integers of maximum size 1000000. For T_2, we generated a random $10 \times 10 \times 200$ array. For T_3, we generated the same order of magnitude of data as for T_2, but in a slightly more intricate record structure. For T_4, we generated a $10 \times 10 \times 10$ array of random integers.

Note that the types tell us that the nesting degree of T_1, T_2, and T_4 is three, and that of T_3 is five. Thus, both our algorithm and Ferry generate three shredded queries for the former and five shredded queries for the latter.

As an example, here is the innermost shredded query generated by our implementation on T_2:

```
  (with q1691 as
    (select (t1677."a") as "1_1_a",
            (t1678."a") as "1_2_a",
            (t1678."b") as "1_2_b",
            (row_number() over (order by t1677."a",t1678."a",t1678."b")) as "2"
     from t1 as t1677,
          t2 as t1678
     where (t1677."a") = (t1678."a"))
   select (2) as "1_1",
          (t1692."2") as "1_2",
          (t1679."v") as "2"
   from q1691 as t1692,
        t3 as t1679
   where (t1692."1_2_b") = (t1679."b"))
union all
  (with q1693 as
    (select (t1680."a") as "1_1_a",
            (t1681."a") as "1_2_a",
            (t1681."b") as "1_2_b",
            (row_number() over (order by t1680."a",t1681."a",t1681."b")) as "2"
     from t1 as t1680,
          t2 as t1681
     where (t1680."a") = (t1681."a"))
   select (5) as "1_1",
          (t1694."2") as "1_2",
          (t1682."v") as "2"
   from q1693 as t1694,
        t4 as t1682
   where (t1694."1_2_b") = (t1682."b"))
```

and the corresponding shredded query generated by Ferry:

```
with
-- binding due to rownum operator
t0000 (item3_int, iter4_nat) as
  (select a0000.a as item3_int,
          row_number () over (order by a0000.a asc) as iter4_nat
     from t1 as a0000),

-- binding due to rownum operator
t0001 (item3_int, iter4_nat, item1_int, item2_int, iter50_nat) as
  (select a0001.item3_int, a0001.iter4_nat, a0002.a as item1_int,
          a0002.b as item2_int,
          row_number () over
          (order by a0001.iter4_nat asc, a0002.a asc, a0002.b asc) as iter50_nat
     from t0000 as a0001,
          t2 as a0002),

-- binding due to rownum operator
t0002 (item3_int, iter4_nat, item1_int, item2_int, iter50_nat,
  item51_bool, item30_int, item31_int, iter52_nat) as
  (select a0003.item3_int, a0003.iter4_nat, a0003.item1_int, a0003.item2_int,
          a0003.iter50_nat,
          case when a0003.item3_int = a0003.item1_int then 1 else 0 end as
          item51_bool, a0004.b as item30_int, a0004.v as item31_int,
          row_number () over
          (order by a0003.iter50_nat asc, a0004.b asc, a0004.v asc) as
          iter52_nat
     from t0001 as a0003,
          t3 as a0004
    where a0003.item3_int = a0003.item1_int),

-- binding due to rownum operator
t0003 (item3_int, iter4_nat, item1_int, item2_int, iter50_nat,
  item51_bool, item18_int, item19_int, iter56_nat) as
  (select a0006.item3_int, a0006.iter4_nat, a0006.item1_int, a0006.item2_int,
          a0006.iter50_nat,
          case when a0006.item3_int = a0006.item1_int then 1 else 0 end as
          item51_bool, a0007.b as item18_int, a0007.v as item19_int,
          row_number () over
          (order by a0006.iter50_nat asc, a0007.b asc, a0007.v asc) as
          iter56_nat
     from t0001 as a0006,
          t4 as a0007
    where a0006.item3_int = a0006.item1_int),

-- binding due to set operation
t0004 (iter42_nat, pos43_nat, item44_int, pos45_nat) as
  ((select a0005.iter50_nat as iter42_nat, a0005.iter52_nat as pos43_nat,
           a0005.item31_int as item44_int, 1 as pos45_nat
      from t0002 as a0005
     where a0005.item3_int = a0005.item1_int
       and a0005.item2_int = a0005.item30_int)
   union all
   (select a0008.iter50_nat as iter42_nat, a0008.iter56_nat as pos43_nat,
           a0008.item19_int as item44_int, 2 as pos45_nat
      from t0003 as a0008
     where a0008.item3_int = a0008.item1_int
       and a0008.item2_int = a0008.item18_int)),

-- binding due to set operation
t0005 (iter13_nat, pos14_nat, item15_nat, pos16_nat) as
  ((select a0010.iter4_nat as iter13_nat, a0010.iter50_nat as pos14_nat,
           a0010.iter50_nat as item15_nat, 2 as pos16_nat
      from t0001 as a0010
     where a0010.item3_int = a0010.item1_int)
   union all
   (select a0011.iter4_nat as iter13_nat, a0011.iter50_nat as pos14_nat,
           a0011.iter50_nat as item15_nat, 1 as pos16_nat
      from t0001 as a0011
     where a0011.item3_int = a0011.item1_int)),

-- binding due to rownum operator
t0006 (iter13_nat, pos14_nat, item15_nat, pos16_nat, pos17_nat) as
  (select a0012.iter13_nat, a0012.pos14_nat, a0012.item15_nat, a0012.pos16_nat,
          row_number () over
          (order by a0012.iter13_nat asc, a0012.pos14_nat asc, a0012.pos16_nat
           asc) as pos17_nat
     from t0005 as a0012)

select a0013.pos17_nat, a0009.item44_int
   from t0004 as a0009,
        t0006 as a0013
  where a0009.iter42_nat = a0013.item15_nat
    and a0013.pos16_nat = a0009.pos45_nat
  order by a0013.pos17_nat asc, a0009.pos43_nat asc
```

The Ferry backend includes some information about where each with-bound subquery comes from. Notice that in this instance Ferry generates five row number operations to our two. Generating row numbers prematurely can be particularly costly, as it can require materialising large amounts of data that could be filtered out.