Comprehending queries over finite maps

Citation for published version:
Ricciotti, W 2023, Comprehending queries over finite maps. in *PPDP'23: Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming.*, 8, ACM International Conference Proceedings Series, ACM Association for Computing Machinery, pp. 1-12, 25th International Symposium on Principles and Practice of Declarative Programming, Lisbon, Portugal, 22/10/23. https://doi.org/10.1145/3610612

Digital Object Identifier (DOI):
10.1145/3610612

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
PPDP'23: Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming

Comprehending queries over finite maps

Wilmer Ricciotti
research@wilmer-ricciotti.net
Laboratory for Foundations of Computer Science, University of Edinburgh
Edinburgh, Scotland, UK

Abstract

Recent programming languages research has developed *language-integrated query*, a convenient technique to seamlessly embed a domain-specific database query language into a general-purpose host programming language; such queries are then automatically converted to the language understood by the target DBMS (e.g. SQL) while at the same time taking advantage of the host language's type-checker to prevent failure at run-time. The embedded query language is often equipped with a rewrite system which normalizes queries to a form that can be directly translated to the DBMS query language.

However, the theoretical foundations of such rewrite systems have not been explored to their full extent, particularly when constructs like grouping and aggregation, which are ubiquitous in real-world database queries, are involved. In this work, we propose an extension of the nested relational calculus with grouping and aggregation which can provide such foundations. Along with strong normalization and translatability to SQL we show that, remarkably, this extension can also blend with *shredding* techniques proposed in the literature to allow queries with a nested relational type to be executed on the DBMS.

CCS Concepts: • **Information systems → Query languages**; • **Software and its engineering → Formal language definitions**.

Keywords: language-integrated query, nested relations, finite maps, multisets

ACM Reference Format:
Wilmer Ricciotti. 2018. Comprehending queries over finite maps. *J. ACM* 37, 4, Article 111 (August 2018), 20 pages. https://doi.org/XXXXXXX.XXXXXXX

Author's address: Wilmer Ricciotti, research@wilmer-ricciotti.net, Laboratory for Foundations of Computer Science, University of Edinburgh, 10 Crichton St, Edinburgh, Scotland, UK, EH8 9AB.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Many, if not all, real-world applications involve interaction with data, often stored and managed in a database system with its own domain-specific language for queries and updates, with relational database management systems (RDBMS) using the SQL standard among the most popular. This architecture has many benefits, particularly separation of concerns: programmers (supposedly) need only express the declarative needs of the application through queries or updates, leaving database implementors free to choose efficient implementation strategies. Nevertheless, database programming can be an unpleasant chore since SQL (for example) presents a rather different interface and abstractions to programmers than most general-purpose languages in which the main application logic is implemented do. Addressing this difficulty has been a major subject of research.

Previous work dating to the late 1980s [31] has established the value of viewing database collection types such as sets and bags as *monads* equipped with operations such as "return" (singleton) and "bind" (concat-map, flat-map, or one-generator comprehension), usually equipped with additional operations such as a zero (empty collection) and plus (union). From a programming languages point of view, this perspective has the distinct advantage of taming queries by making them an instance of a well-understood and explored interface (particularly in languages such as Haskell that directly support monads via type classes). And indeed, there has now been considerable work exploiting this correspondence, which began in earnest with Wong's Kleisli system, and has subsequently informed systems such as Microsoft's LINQ for SQL (particularly the F# implementation), the Links cross-tier web programming language, and libraries such as Quill for Scala.

Despite the success of this general approach, dark corners remain, due to the fact that query languages such as SQL are considerably more restricted than general-purpose programming languages, particularly those those with higher-order functions such as F#, Scala or Links. The free combination of type constructors such as collection and record types is a given in most programming languages, but such *nested collections* are not normally supported in RDBMSs (though this facility is supported by the SQL:2003 standard, it is not widely adopted). Similarly, it seems natural to use higher-order functions to decompose query expressions into smaller, reusable and more readable, chunks, but SQL does not support higher-order functions (and even using user-defined

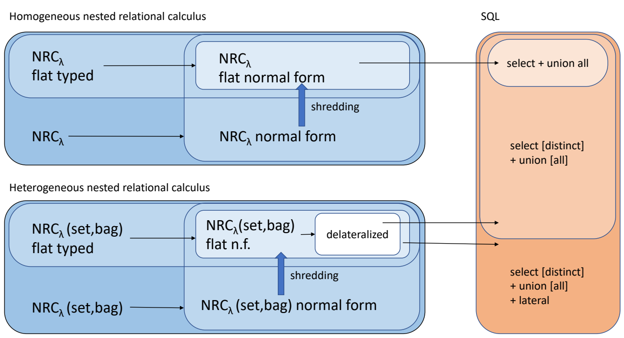

Figure 1. Relationship between fragments of $\mathcal{NRC}$ and of SQL.

first-order functions inside queries can have a high cost). As a result, some previous work has considered the following problems: How can we identify which host language query expressions can actually be performed on the database? When features that seem natural in the host language are not directly supported by the database, can these nevertheless be simulated somehow? Proposals addressing the first question include type-and-effect systems (as in Links [15]) or staged calculi (for F# [3]) to identify query expressions that can safely be performed on the database, while the second has been addressed in part by developing query normalization techniques that simplify queries that employ nonrecursive higher-order functions and *shredding* transformations that simulate queries over nested data using queries over flat data (that is, queries that manipulate only collections of records of primitive values, as SQL requires.)

Core calculi like the nested relational calculus ($\mathcal{NRC}$) [2] offer an elegant way to describe the monadic sublanguage that is mutually understandable to a general purpose language and SQL and have been employed to study query normalization and shredding in the presence of non-recursive higher-order functions [4–6]. Such techniques were initially investigated for bag semantics only, but have been since extended to work with queries mixing set and bag semantics [21, 22, 24].

Figure 1 shows the relationship between fragments of two variants of $\mathcal{NRC}$ and fragments of SQL. In the homogeneous $\mathcal{NRC}$, using bag semantics, terms whose type is a flat relation can be normalized and directly translated to SQL queries employing `SELECT` and `UNION ALL`. Terms with a nested relational type eventually rewrite to nested relational forms that can be converted to multiple flat relational normal forms by shredding, which can be translated to SQL as described above.

The heterogeneous calculus (also termed $\mathcal{NRC}_\lambda(\mathit{Set}, \mathit{Bag})$) adds the ability to mix sets and bags in the same query. Just like in the homogeneous variant, every term can be normalized and normal forms with a nested relational type can be shredded to flat normal forms. Predictably, these can be translated to a larger SQL fragment including `SELECT DISTINCT` and `UNION` operations; however, generally we will also need to use `LATERAL` joins (available since SQL:1999); an optional delateralization step converts $\mathcal{NRC}_\lambda(\mathit{Set}, \mathit{Bag})$ normal forms to a form that can be translated to SQL without using `LATERAL`. Notice that the heterogeneous calculus, including its rewrite system, is a superset of the homogeneous one, therefore terms of this calculus only employing bag semantics are guaranteed to be translated to the same SQL queries produced by the homogeneous calculus.

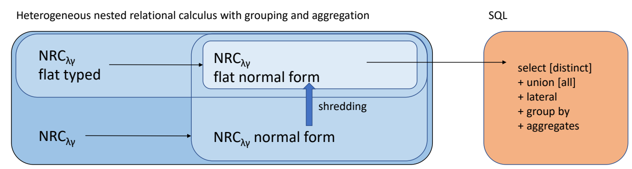

Figure 2. Extending $\mathcal{NRC}$ to target SQL with grouping.

Another area where improvement is needed, and which is the focus of this paper, is the coverage of the most important SQL features. In particular, so-called OLAP or *on-line analytic processing* queries rely on *grouping and aggregation* capabilities of SQL. The importance of these capabilities is highlighted by their use in 16 out of 22 of the TPC-H benchmark queries ([30]), which are by far the most widely used benchmark for SQL query processing and optimization. We want to eventualy make language-integrated query "TPC-H complete", or in other words capable of handling (in a principled way) all of the features needed for the TPC-H queries. As a simple concrete example, suppose we have an employee database and we wish to calculate the average salary in each department. In SQL this can be done as follows:

```
q_group := SELECT d.name,AVERAGE(e.salary)
           FROM department d, employee e
           WHERE d.id = e.dept
           GROUP BY d.name
```

We intend to support grouping and aggregation in a functional programming language by means of further extensions to $\mathcal{NRC}$: the intended operation of such an extension is described by Figure 2. We start with minimal expectations concerning the operations that should be supported; since our goal is to allow idiomatic functional programs to be converted to SQL queries, these operations need not mimic SQL queries closely, but it is important for them to provide a natural interface for the functional programmer. Our initial proposal is based on the understanding that grouping and aggregation are informally seen as separate operations, and the reason why SQL requires grouping to be associated with aggregation (aside from pathological cases) is based on its limited type system, where all queries must evaluate to finite collections of records of scalars (also termed *(flat) relations*).

Operationally, grouping takes a relation and partitions it into a number of subrelations, each of which is indexed by a key for that group. While a way to represent such an indexing would be by means of a function type, general functions cannot be represented as relations, meaning such types would not be a suitable source for a translation to SQL. However, the maps resulting from grouping associate the elements of a *finite* domain set to finite relations (i.e. finite collections): we call these maps *finite maps*. If we represent multiset collections with element type T as $[T]$ and finite maps from records of type ρ to multiset collections with element type T as $[T]_\rho$, we can see that finite maps can be represented as SQL relations by employing the isomorphism:

$$[T]_\rho \simeq [\rho \times T]$$

Now we tentatively list the operations that a language with finite maps, grouping, and aggregation should support:

- given a collection l of type $[T]$ and an indexing function f assigning a key of record type ρ to any value of type T, $groupBy\ f\ l$ should return a finite map associating each $k : \rho$ to the collection of those values v in l for which k is equal to the key computed from v, i.e. $f\ v$:
$$groupBy : (T \to \rho) \to [T] \to [T]_\rho$$
- a lookup operation which, given a finite map m of type $[T]_\rho$ and a record k of type ρ, returns the collection associated by m to the key k:
$$lookup : [T]_\rho \to \rho \to [T]$$
- an aggregation function α takes a collection of type $[S]$ and returns a value of type T (the language will enforce the property that such aggregation functions should be expressible in SQL); but for the very common case in which aggregation is performed on the result of grouping (i.e. on a finite map), we should be able to express an operation *aggBy* to lift aggregations from collections to indexed collections:
$$aggBy : ([T] \to T) \to [S]_\rho \to [\rho \times T]$$
- Crucially, we will also need to extend comprehension from collections to finite maps; we can come up with several options, which we distinguish by annotating the comprehension generator by different superscripts.

Perhaps the most obvious solution is to provide comprehension for finite maps by implicitly converting them to their isomorphic collections: we denote this operation with $\stackrel{\mathcal{M}}{\leftarrow}$ generators

$$\left[R \mid (k, x) \stackrel{\mathcal{M}}{\leftarrow} M\right]$$

This should take the disjoint unions of all the R evaluated for each key k of M and each value x in the collection returned by $lookup\ M\ k$. Since the values that M maps to k are considered separately rather than as a collection, this form of comprehension is not very natural. Alternatively, we can consider a generator returning keys k together with the collection v of all values associated to k by M: we use $\stackrel{\mathcal{G}}{\leftarrow}$ generators to denote this kind of comprehension:

$$\left[R \mid (k, v) \stackrel{\mathcal{G}}{\leftarrow} M\right]$$

Here, each key k will be returned only once, together with a v which will equal $lookup\ M\ k$; since v can be computed from M and k, we can even drop it and consider generators returning only the keys of a given finite map, which we denote by $\stackrel{\mathcal{K}}{\leftarrow}$:

$$\left[R \mid k \stackrel{\mathcal{K}}{\leftarrow} M\right]$$

While we could consider a language employing any of these comprehensions (potentially expressing the other two as derived operators), the queries we express will have to be converted to SQL. In order for this to happen we will need to provide a rewrite system to convert the more liberal functional queries into normal forms that can be easily translated to SQL.

2 A nested relational calculus with finite maps

2.1 Syntax

We introduce an extension of the heterogeneous nested relational calculus $\mathcal{NRC}_\lambda(Set, Bag)$ [21] with finite maps: we will call the new calculus $\mathcal{NRC}_{\lambda\gamma}$. Like its predecessor, $\mathcal{NRC}_{\lambda\gamma}$ allows the mixing of two different kinds of collections: sets and bags (also known as multisets). The main extension of the new calculus consists in a *grouping* operator γ which converts collections into maps assigning collections to *keys* in a certain finite domain; these finite maps can be seen as a generalization of collections and thus collection operators such as unions and comprehensions are extended to finite maps. The calculus also supports aggregation operators from bags of values to simple values.

In our previous work we provided two different versions for most collection operators depending on whether they applied to sets or bags (for instance: $\cup$ for set union, and $\uplus$ for disjoint bag union): this created a large syntactic overhead by requiring us to consider all of these operators twice. In the syntax below, instead, we factorize such operator pairs by using the same symbol (say $+\!\!+$ for union over any collection), and discriminate between the two cases by annotating said operator with its collection kind $\mathcal{S}$ = set or bag (i.e. $+\!\!+^{\mathrm{set}}, +\!\!+^{\mathrm{bag}}$). We will often omit the annotation when it is irrelevant or can be easily inferred from the context.

$$
\begin{array}{lcl}
\textbf{Types:} & & \\
\mathcal{S} & ::= & \mathsf{set} \mid \mathsf{bag} \\
S, T & ::= & b \mid S \rightarrow T \mid \langle P \rangle \mid [T]^{\mathcal{S}}_{P} \\
P & ::= & \overrightarrow{\ell : T} \\
& & \\
\textbf{Terms:} & & \\
L, M, N & ::= & x \mid t \mid c(\overrightarrow{M}) \mid \langle \overrightarrow{\rho} \rangle \mid M.\ell \\
& \mid & \lambda x.M \mid M\ N \mid \alpha(M) \mid M \ \mathbf{where}^{\mathcal{S}}\ N \\
& \mid & []^{\mathcal{S}} \mid [\rho \triangleright M]^{\mathcal{S}} \mid M \uplus^{\mathcal{S}} N \mid [M \mid x \leftarrow N]^{\mathcal{S}} \\
& \mid & \gamma^{\mathcal{S}}_{x.\rho}(M) \mid M \circledast^{\mathcal{S}} N \mid \left[M \mid k \overset{\mathcal{K}}{\leftarrow} N \right]^{\mathcal{S}} \\
& \mid & \delta M \mid \iota M \mid aggBy^{\mathcal{S}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}}(M) \\
\rho & ::= & \overrightarrow{\ell = M} \\
\alpha & ::= & \mathbf{count} \mid \mathbf{sum} \mid \mathbf{min} \mid \mathbf{max} \mid \mathbf{avg}
\end{array}
$$

$\mathcal{NRC}_{\lambda\gamma}$ includes refined types $[T]^{\mathsf{set}}_{P}$ and $[T]^{\mathsf{bag}}_{P}$ representing respectively set-valued and bag-valued finite maps: terms of these types behave as functions whose domain is described by a *row type* P (where a row type is defined as a collection of items in the form $\ell : T'$, such that ℓ is a label and T' a regular type) and whose codomain is either a set or a bag of values of the target type T; such maps are *finite* because they will return a non-empty collection only for a finite subset of their domain. When P is an empty row, we obtain the traditional types $[T]^{\mathsf{set}}$ and $[T]^{\mathsf{bag}}$ of sets and bags over an object type T. Row types are also used to express record types $\langle P \rangle$. The grammar of types is completed by atomic types b (which must include Booleans $\mathbf{B}$ and numbers $\mathbf{N}$) and function types $S \rightarrow T$.

The terms allow many standard forms including variables x, applied constants $c(\overrightarrow{M})$, function abstraction and application ($\lambda x.M$ and $M\ N$). Given a row ρ consisting of items in the form $\ell = M$ associating a term to a label, $\langle \rho \rangle$ represents the record containing that row. As usual, $M.\ell$ is used to access field ℓ in a record M.

Maps include terms that are similar to the collection terms in $\mathcal{NRC}_{\lambda}(Set, Bag)$, but with some differences: while the empty map $[]^{\mathcal{S}}$ and union map $M \uplus^{\mathcal{S}} N$ are syntactically indistinguishable from the corresponding concepts of $\mathcal{NRC}_{\lambda}(Set, Bag)$, the singleton map $[\rho \triangleright M]^{\mathcal{S}}$ represents an expression associating the record $\langle \rho \rangle$ to the singleton collection $[M]^{\mathcal{S}}$ – similarly to what we did with map types, we identify $[M]^{\mathcal{S}}$ with the singleton map $[\cdot \triangleright M]^{\mathcal{S}}$ having the empty row $\cdot$ as its source. The operation $[M \mid x \leftarrow N]^{\mathcal{S}}$ is also syntactically identical to comprehension in previous versions of $\mathcal{NRC}$, but as we will see later its typing rule is relaxed; we will call this operation *value comprehension map* to distinguish it from a *key* comprehension operation that we will introduce soon. The one armed conditional $M\ \mathbf{where}^{\mathcal{S}}\ N$ evaluating to M or $[]^{\mathcal{S}}$ depending on whether the Boolean N is true or false is also unsurprising.

All of the aforementioned term forms can be applied to either collection kind; two operations that defy this scheme are deduplication δM (which converts a bag-valued map to a set-valued one by deduplicating all of its outputs) and promotion ιM (performing the inverse opeation by converting the set outputs of the map M to bags containing the exact same elements, with multiplicity equal to one).

So far, the terms of $\mathcal{NRC}_{\lambda\gamma}$ are very similar to those of $\mathcal{NRC}_{\lambda}(Set, Bag)$, although with refined typing and semantics; however, $\mathcal{NRC}_{\lambda\gamma}$ also introduces new operations to manipulate maps that have no counterpart in $\mathcal{NRC}_{\lambda}(Set, Bag)$: the simplest of these is the lookup operation $M \circledast^{\mathcal{S}} N$ which, given a map M and a key N, returns the value of M associated to N. Since a finite map M only associates an output to a finite number of keys N, but $\mathcal{NRC}_{\lambda\gamma}$ allows its types to have infinite inhabitants, it may happen that N is not in the domain of M: in this case, the semantics of $M \circledast^{\mathcal{S}} N$ is the empty collection.

The grouping operation $\gamma^{\mathcal{S}}_{x.\rho}(M)$ binds the variable x in the row ρ: if M is a collection (a pure set or bag), for each occurrence of an element N in M, the grouping operation produces a singleton map sending the row $\rho\,[N/x]$ to N; if M is itself a proper map rather than a pure collection, the behaviour is similar, but the rows resulting from ρ are combined with those already present in M.

The groupwise aggregation $aggBy^{\mathcal{S}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}}(M)$ takes as its main input M, which must be a finite map whose target type is a collection of records of base type (or, for short, a *flat relational map*); the secondary input is an aggregation clause using a bound variable z (used to reference the target of the map M), arbitrary output attributes $\overrightarrow{\ell}$, and input attributes $\overrightarrow{\ell'}$ matching attributes of the relation M.

For example, asssume $M_{factors} : [\langle \mathsf{f} : \mathbf{N} \rangle]^{\mathsf{bag}}_{\mathsf{num}:\mathbf{N}}$ is a bag-valued relational map associating integers smaller than 100 to the bag of their prime factors, with their multiplicity: so for example $\langle \mathsf{num} = 2 \rangle$ is mapped to the bag $[\langle \mathsf{f} = 2 \rangle]^{\mathsf{bag}}$, $\langle \mathsf{num} = 12 \rangle$ to the bag $[\langle \mathsf{f} = 2 \rangle, \langle \mathsf{f} = 2 \rangle, \langle \mathsf{f} = 3 \rangle]^{\mathsf{bag}}$, and $\langle \mathsf{num} = 99 \rangle$ to the bag $[\langle \mathsf{f} = 3 \rangle, \langle \mathsf{f} = 3 \rangle, \langle \mathsf{f} = 11 \rangle]^{\mathsf{bag}}$. Then, by using the row $(\mathsf{small} = x \leq 10)$ (which evaluates to true for integers x less than 10, and to false otherwise), we can construct a map $\gamma_{x.\mathsf{small} = x \leq 10}(M_{factors})$ which maps $\langle \mathsf{num} = 12, \mathsf{small} = \mathsf{true} \rangle$ to $[\langle \mathsf{f} = 2 \rangle, \langle \mathsf{f} = 2 \rangle, \langle \mathsf{f} = 3 \rangle]^{\mathsf{bag}}$, $\langle \mathsf{num} = 12, \mathsf{small} = \mathsf{false} \rangle$ to $[]^{\mathsf{bag}}$, $\langle \mathsf{num} = 99, \mathsf{small} = \mathsf{true} \rangle$ to $[\langle \mathsf{f} = 3 \rangle, \langle \mathsf{f} = 3 \rangle]^{\mathsf{bag}}$ and $\langle \mathsf{num} = 99, \mathsf{small} = \mathsf{false} \rangle$ to $[\langle \mathsf{f} = 11 \rangle]^{\mathsf{bag}}$. Groupwise aggregation can be used to count the number of prime factors of each number in the map: the map resulting from $aggBy_{\mathsf{z.c} = \mathbf{count}(\mathsf{z.f})}(M_{factors})$ maps 2 to the singleton $[\langle \mathsf{c} = 1 \rangle]^{\mathsf{bag}}$, and 12 to $[\langle \mathsf{c} = 3 \rangle]^{\mathsf{bag}}$. Notice that since the output type of a finite map is always a collection, the use of *aggBy* results in a map returning singletons rather than naked values.

The final operation is $\left[M \mid k \overset{\mathcal{K}}{\leftarrow} N\right]^{\mathcal{S}}$, which we will call *key* comprehension: it is semantically equivalent to performing a set comprehension over the keys of the map N returning, for each key, a map M where the actual key has been substituted for the variable k. Notice that even in the bag case $\left[M \mid k \overset{\mathcal{K}}{\leftarrow} N\right]^{\mathsf{bag}}$, each key of N is only considered once, since there does not seem to be much use to the concept of multiplicity of the input of a finite map.

To make the syntax of comprehensions less pedantic, we will allow ourselves to use placeholder symbols $\diamond$ and $\clubsuit$ in comprehension generators like $\left[M \mid x \overset{\diamond}{\leftarrow} N\right]^{\mathsf{set}}$ or $\left[M \mid x \overset{\clubsuit}{\leftarrow} N\right]^{\mathsf{bag}}$ to mean either value or key comprehension. Finally, we will use the syntactic sugar $\left[M \mid x_1 \overset{\diamond}{\leftarrow} N_1, \dots, x_n \overset{\clubsuit}{\leftarrow} N_m\right]$, defined as $\left[\cdots\left[M \mid x_n \overset{\clubsuit}{\leftarrow} N_m\right]\cdots \mid x_1 \overset{\diamond}{\leftarrow} N_1\right]$ (and similarly for bags).

2.2 Typing rules

The type system for $\mathcal{NRC}_{\lambda\gamma}$ is given in Figure 3. The rules for variables, records, projections, functions, and applications are standard. A fixed signature Σ assigns to constants and aggregate operators their type: the former take a sequence of arguments of scalar type and return a scalar; the latter take as input a single bag of scalar-typed items and return a scalar.

The typing rules for finite maps are largely based on those of $\mathcal{NRC}_\lambda(Set, Bag)$ collections, but with important differences. Empty maps $[]$ are allowed at any finite map type, whereas singletons have a map type corresponding to their arguments: if ρ has a row type P and M has type T, then $[\rho \triangleright M]^{\mathcal{S}}$ has type $[T]_P^{\mathcal{S}}$. Unions combine collections preserving the type of their arguments (which must be the same).

The typing rules for value comprehension are more interesting: to construct the term $[M \mid x \leftarrow N]^{\mathcal{S}}$, we will require the generator N to be a pure collection of type $[T]^{\mathcal{S}}$; then the output term M, existing in an extended context where x gets values of type T, can have any set-valued map type $[S]_P^{\mathsf{set}}$; the result will have the same type as M – it must be stressed that this is a significant extension over $\mathcal{NRC}_\lambda(Set, Bag)$, as in that calculus M (and the comprehension output) would need to be pure collections. Deduplication and promotion convert a bag-valued map type into the corresponding set-valued map type and vice-versa; one-armed conditionals are similar as their $\mathcal{NRC}$ version, except for the fact that their output can be a map rather than a pure collection.

As for key comprehensions $\left[M \mid k \overset{\mathcal{K}}{\leftarrow} N\right]^{\mathcal{S}}$, the generator N can of course be a proper map of type $[T]_{P'}^{\mathcal{S}}$ (as it would otherwise be pointless to perform comprehension over its keys), and the output M once again must have a $\mathcal{S}$-valued map type when evaluated in an extended context: however, this time the type of the bound variable k must be a tuple over the row type P.

Grouping $\gamma_{x.\rho}(M)$ adds new keys to the map M: if M has type $[T]_P$ and ρ has row type P' in the extended context where $x : T$, then the whole term has type $[T]_{P \oplus P'}$, where $P \oplus P'$, only defined when P and P' have disjoint domains, contains exactly those label-type associations that appear in either P or P'; should P and P' have overlapping label sets, the typechecker will reject the term. The lookup operation $M \circledast N$ requires M to have a map type $[T]_P$ and N to be a tuple over the row type P (matching the expected input of M), and returns the pure set of type $[T]$ associated to that input. Notice that, in order to syntactically avoid unnecessary "detour" lookup operations, we require that the lookup should only happen on proper finite maps rather than pure collections: therefore, if M is a pure collection and N has an empty tuple type, $M \circledast N$ is not syntactically well-formed; this is not a limitation because the expected semantics of that term would be the same as that of M.

Groupwise aggregation $aggBy_{\overrightarrow{z.\ell_n = \alpha_n(z.\ell'_n)}}(M)$ assumes that M has type $[\langle P' \rangle]_P$: for each key k of M and for each tuple z in $M \circledast k$, $aggBy$ will produce a key-value pair where the key is k (as it was in the input) and the value is a singleton tuple containing fields $\ell_i = \alpha_i(\ell'_i)$ for $i = 1, \dots, n$. The types of the fields in the codomain of map M must match the input of the aggregate functions using them (meaning that, for each $\alpha_i(z.\ell'_i)$, if the type of α_i is $[b_i]^{\mathsf{bag}} \Rightarrow b'_i$, then the row type P' must associate the label ℓ'_i to the b_i matching the input of α_i, where $i = 1, \dots, n$). The resulting type is again a finite map with the same domain as that of its input M; however each output of the map will be a singleton tuple (due to the use of aggregation) whose fields are $\ell_i : b'_i$ determined by the aggregation clause.

While value comprehension, grouping, lookup and groupwise aggregation have similar typing rules for both collection kinds, key comprehension is less obvious. The reason for this is that in a bag-valued finite map, the output provides multiplicity information, but there is no such information for its input. Consequently, the typing rule for key comprehensions returning bag-valued maps still requires the generator to be a set-valued map; however it is always possible to deduplicate a bag-valued map and perform key comprehension on the resulting set-valued map.

Example 2.1. The query `q_group` from the introduction can be expressed in $\mathcal{NRC}_{\lambda\gamma}$ by a combination of comprehension, singleton finite maps, and groupwise aggregation:

$$M_{group} := \quad aggBy_{z.\mathsf{salary}=\mathbf{avg}(z.\mathsf{salary})} \left(\begin{array}{l} \Big[\Big[\mathsf{dept} = d.\mathsf{name} \\ \quad \triangleright \langle \mathsf{salary} = e.\mathsf{salary} \rangle\Big] \\ \quad \mathbf{where}\ (d.\mathsf{id} = e.\mathsf{dept}) \\ \mid d \leftarrow \mathit{department}, e \leftarrow \mathit{employee}\Big] \end{array} \right)$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Sigma(c) = \overrightarrow{b_n} \Rightarrow b' \qquad (\Gamma \vdash M_i : b_i)_{i=1,\dots,n}}{\Gamma \vdash c(\overrightarrow{M_n}) : b'} \qquad \frac{\Sigma(\alpha) = [b]^{\mathrm{bag}} \Rightarrow b' \qquad \Gamma \vdash M : [b]^{\mathrm{bag}}}{\Gamma \vdash \alpha(M) : b'}$$

$$\frac{\vec{\ell} \text{ is duplicate-free} \qquad (\Gamma \vdash M_i : T_i)_{i=1,\dots,n}}{\Gamma \vdash \overrightarrow{\ell_n = M_n} : \overrightarrow{\ell_n : T_n}} \qquad \frac{\Gamma \vdash \rho : P}{\Gamma \vdash \langle \rho \rangle : \langle P \rangle} \qquad \frac{\Gamma \vdash M : \langle P \rangle \qquad \ell \in dom(P)}{\Gamma \vdash M.\ell : P(\ell)}$$

$$\frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x.M : S \to T} \qquad \frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash (M\ N) : T} \qquad \frac{\Gamma \vdash M : [T]_P^{\mathcal{S}} \qquad \Gamma \vdash N : \mathbf{B}}{\Gamma \vdash M\ \mathbf{where}^{\mathcal{S}}\ N : [T]_P^{\mathcal{S}}}$$

$$\frac{}{\Gamma \vdash []^{\mathcal{S}} : [T]_P^{\mathcal{S}}} \qquad \frac{\Gamma \vdash \rho : P \qquad \Gamma \vdash M : T}{\Gamma \vdash [\rho \triangleright M]^{\mathcal{S}} : [T]_P^{\mathcal{S}}} \qquad \frac{\Gamma \vdash M : [T]_P^{\mathcal{S}} \qquad \Gamma \vdash N : [T]_P^{\mathcal{S}}}{\Gamma \vdash M +\!\!\!+^{\mathcal{S}} N : [T]_P^{\mathcal{S}}} \qquad \frac{\Gamma, x : T \vdash M : [S]_P^{\mathcal{S}} \qquad \Gamma \vdash N : [T]^{\mathcal{S}}}{\Gamma \vdash [M \mid x \leftarrow N] : [S]_P^{\mathcal{S}}}$$

$$\frac{\Gamma \vdash M : [T]_P^{\mathrm{bag}}}{\Gamma \vdash \delta M : [T]_P^{\mathrm{set}}} \qquad \frac{\Gamma \vdash M : [T]_P^{\mathrm{set}}}{\Gamma \vdash \iota M : [T]_P^{\mathrm{bag}}} \qquad \frac{\Gamma \vdash M : [T]_P^{\mathcal{S}} \qquad \Gamma, x : T \vdash \rho : P' \qquad \rho \text{ not empty}}{\Gamma \vdash \gamma_{x.\rho}^{\mathcal{S}}(M) : [T]_{P \oplus P'}^{\mathcal{S}}} \qquad \frac{\Gamma \vdash M : [T]_P^{\mathcal{S}} \qquad \Gamma \vdash N : \langle P \rangle \qquad P \text{ not empty}}{\Gamma \vdash M \circledast^{\mathcal{S}} N : [T]^{\mathcal{S}}}$$

$$\frac{\Gamma, x : \langle P' \rangle \vdash M : [S]_P^{\mathcal{S}} \qquad \Gamma \vdash N : [T]_{P'}^{\mathrm{set}}}{\Gamma \vdash \left[M \mid k \overset{\mathcal{K}}{\leftarrow} N\right]^{\mathcal{S}} : [S]_P^{\mathcal{S}}} \qquad \frac{\Gamma \vdash M : \left[\overrightarrow{\langle P' \rangle}\right]_P^{\mathcal{S}} \qquad (\Sigma(\alpha_i) = [b_i]^{\mathrm{bag}} \Rightarrow b'_i)_{i=1,\dots,n} \qquad (P(\ell'_i) = b_i)_{i=1,\dots,n}}{\Gamma \vdash aggBy^{\mathcal{S}}_{\overrightarrow{z.\ell_n = \alpha_n(z.\ell'_n)}}(M) : \left[\langle \overrightarrow{\ell_n : b'_n} \rangle\right]_P^{\mathcal{S}}}$$

Figure 3. Type system of $\mathcal{NRC}_{\lambda\gamma}$.

2.3 Rewrite system

In Figure 4 we give a rewrite system to normalize $\mathcal{NRC}_{\lambda\gamma}$ queries. These include beta reduction for applied functions and a similar rule for projection on a record literal. We import from $\mathcal{NRC}_{\lambda}(Set, Bag)$ all of the normalization rules involving collections (although in this calculus they will be applied to finite maps); among those rules, we note the beta-like rule for contracting value comprehensions whose generators are singletons and the "associativity" rules for unnesting value comprehensions:

$$\left[M \mid x \leftarrow [N]^{\mathcal{S}}\right]^{\mathcal{S}} \rightsquigarrow M\,[{}^{N}\!/_{x}]$$
$$\left[L \mid y \leftarrow [N \mid x \leftarrow M]^{\mathcal{S}}\right]^{\mathcal{S}} \rightsquigarrow [L \mid x \leftarrow M, y \leftarrow N]^{\mathcal{S}}$$

with the side condition that, to prevent variable capture, the second rule can only be applied if x does not appear free in L.

We now wonder whether similar rules would work for key comprehension: the beta rule needs to be adapted to use the key in the singleton (rather than the value) as the variable replacement:

$$\left[M \mid k \overset{\mathcal{K}}{\leftarrow} [\rho \triangleright N]\right] \rightsquigarrow M\,[{}^{\langle\rho\rangle}\!/_{k}]$$

As for unnesting, we have an interesting situation: for set-valued maps, any combination of value and key comprehension can be unnested:

$$\left[L \mid y \overset{\clubsuit}{\leftarrow} \left[N \mid x \overset{\diamond}{\leftarrow} M\right]^{\mathrm{set}}\right]^{\mathrm{set}} \rightsquigarrow \left[L \mid x \overset{\diamond}{\leftarrow} M, y \overset{\clubsuit}{\leftarrow} N\right]^{\mathrm{set}}$$

where $\diamond$ and $\clubsuit$ can stand for value or key comprehension, indifferently; on the other hand, when we use bag-valued maps, we are unable to unnest a comprehension out of a key comprehension

$$\left[L \mid y \overset{\mathcal{K}}{\leftarrow} \left[N \mid x \overset{\diamond}{\leftarrow} M\right]^{\mathrm{set}}\right]^{\mathrm{bag}} \not\rightsquigarrow \left[L \mid x \overset{\diamond}{\leftarrow} M, y \overset{\mathcal{K}}{\leftarrow} N\right]^{\mathrm{bag}}$$

Recall that the generator of a key comprehension returning a bag-valued map must be a set-valued map: semantically there is an implicit promotion from set to bag, preventing unnesting from being sound. So, a key-comprehension returning a bag-valued map behaves similarly to a value-comprehension whose generator is in the form ιN: even if N is a comprehension, it is wrapped in a ι operation which blocks unnesting.

On the other hand, unnesting a comprehension of either kind out of a value comprehension is semantically sound,

$$
\begin{array}{c}
(\lambda x.M)\ N \leadsto M[N/x] \qquad \langle \ldots, \ell = M, \ldots \rangle.\ell \leadsto M \\[1em]
\begin{array}{rll}
\gamma_{x.\rho}([\,]) \leadsto [\,] \qquad \gamma_{x.\rho}([\rho' \triangleright M]) \leadsto [\rho\,[M/x] \oplus \rho' \triangleright M] & & \\
\gamma_{x.\rho}(M +\!\!+ N) \leadsto \gamma_{x.\rho}(M) +\!\!+ \gamma_{x.\rho}(N) \qquad \gamma_{x.\rho}(\gamma_{x.\rho'}(M)) \leadsto \gamma_{x.\rho\oplus\rho'}(M) & & \\
\gamma_{x.\rho}(\left[L \mid y \xleftarrow{\diamond} M\right]) \leadsto \left[\gamma_{x.\rho}(L) \mid y \xleftarrow{\diamond} M\right] & & (\text{if } y \notin FV(x.\rho)) \\
\gamma_{x.\rho}(M \ \mathbf{where}\ N) \leadsto \gamma_{x.\rho}(M)\ \mathbf{where}\ N & & \\
[\,] \circledast N \leadsto [\,] \qquad (L +\!\!+ M) \circledast N \leadsto (L \circledast N) +\!\!+ (M \circledast N) & & \\
\left[L \mid x \xleftarrow{\diamond} M\right] \circledast N \leadsto \left[L \circledast N \mid x \xleftarrow{\diamond} M\right] & & (\text{if } x \notin FV(N)) \\
(L\ \mathbf{where}\ M) \circledast N \leadsto (L \circledast N)\ \mathbf{where}\ M & & \\
[\rho \triangleright M] \circledast N \leadsto [M]\ \ \mathbf{where}\ (\langle\rho\rangle = N) & &
\end{array} \\[1em]
\begin{array}{rll}
[\,] +\!\!+ M \leadsto M \qquad M +\!\!+ [\,] \leadsto M \qquad \left[[\,] \mid x \xleftarrow{\diamond} M\right] \leadsto [\,] \qquad \left[M \mid x \xleftarrow{\diamond} [\,]\right] \leadsto [\,] & & \\
[M \mid x \leftarrow [N]] \leadsto M\,[N/x] \qquad \left[M \mid k \xleftarrow{\mathcal{K}} [\rho \triangleright N]^{\text{set}}\right] \leadsto M\,[\langle\rho\rangle/k] & & \\
\left[M +\!\!+ N \mid x \xleftarrow{\diamond} R\right] \leadsto \left[M \mid x \xleftarrow{\diamond} R\right] +\!\!+ \left[N \mid x \xleftarrow{\diamond} R\right] & & \\
\left[M \mid x \xleftarrow{\diamond} N +\!\!+ R\right]^{\text{set}} \leadsto \left[M \mid x \xleftarrow{\diamond} N\right]^{\text{set}} +\!\!+ \left[M \mid x \xleftarrow{\diamond} R\right]^{\text{set}} & & \\
[M \mid x \leftarrow N +\!\!+ R]^{\text{bag}} \leadsto [M \mid x \leftarrow N]^{\text{bag}} +\!\!+ [M \mid x \leftarrow R]^{\text{bag}} & & \\
\left[L \mid y \xleftarrow{\clubsuit} \left[N \mid x \xleftarrow{\diamond} M\right]^{\text{set}}\right]^{\text{set}} \leadsto \left[L \mid x \xleftarrow{\diamond} M, y \xleftarrow{\clubsuit} N\right]^{\text{set}} & & (\text{if } x \notin FV(L)) \\
\left[L \mid y \leftarrow \left[N \mid x \xleftarrow{\diamond} M\right]^{\text{bag}}\right]^{\text{bag}} \leadsto \left[L \mid x \xleftarrow{\diamond} M, y \leftarrow N\right]^{\text{bag}} & & (\text{if } x \notin FV(L)) \\
\left[M \mid x \xleftarrow{\diamond} N\ \mathbf{where}\ R\right] \leadsto \left[M \mid x \xleftarrow{\diamond} N\right]\ \ \mathbf{where}\ R & & \\
\delta[\,]^{\text{bag}} \leadsto [\,]^{\text{set}} \qquad \delta\,[\rho \triangleright M]^{\text{bag}} \leadsto [\rho \triangleright M]^{\text{set}} \qquad \delta(M +\!\!+ N) \leadsto \delta M +\!\!+ \delta N \qquad \delta\gamma_{x.\rho}(M) \leadsto \gamma_{x.\rho}(\delta M) & & \\
\delta\left[M \mid x \xleftarrow{\diamond} N\right]^{\text{bag}} \leadsto \left[\delta M \mid x \xleftarrow{\diamond} \delta N\right]^{\text{set}} \qquad \delta(M\ \mathbf{where}^{\text{bag}}\ N) \leadsto \delta M\ \mathbf{where}^{\text{set}} N & & \\
\iota[\,]^{\text{set}} \leadsto [\,]^{\text{bag}} \qquad \iota\,[\rho \triangleright M]^{\text{set}} \leadsto [\rho \triangleright M]^{\text{bag}} \qquad \iota(M\ \mathbf{where}^{\text{set}}\ N) \leadsto \iota M\ \mathbf{where}^{\text{bag}} N \qquad \delta\iota M \leadsto M & &
\end{array} \\[1em]
\begin{array}{rll}
M\ \mathbf{where}\ \mathsf{true} \leadsto M \qquad M\ \mathbf{where}\ \mathsf{false} \leadsto [\,] & & \\
[\,]\ \mathbf{where}\ M \leadsto [\,] & & \\
(N +\!\!+ R)\ \mathbf{where}\ M \leadsto (N\ \mathbf{where}\ M) +\!\!+ (R\ \mathbf{where}\ M) & & \\
\left[N \mid x \xleftarrow{\diamond} R\right]\ \ \mathbf{where}\ M \leadsto \left[N\ \mathbf{where}\ M \mid x \xleftarrow{\diamond} R\right] & & (\text{if } x \notin FV(M)) \\
(R\ \mathbf{where}\ M)\ \mathbf{where}\ N \leadsto R\ \mathbf{where}\ (M \land N) & &
\end{array}
\end{array}
$$

Figure 4. Normalization rules for $\mathcal{NRC}_{\lambda\gamma}$

hence we can state the rewrite rule:

$$\left[L \mid y \leftarrow \left[N \mid x \xleftarrow{\diamond} M\right]^{\text{bag}}\right]^{\text{bag}} \leadsto \left[L \mid x \xleftarrow{\diamond} M, y \leftarrow N\right]^{\text{bag}}$$

For similar reasons, in the set-valued case, key comprehension distributes over unions both in head and in generator position, just like value comprehension, but in the bag-valued case, key comprehension only distributes over disjoint unions in head position:

$$
\begin{array}{l}
\left[L +\!\!+ M \mid x \xleftarrow{\diamond} N\right]^{\mathcal{S}} \leadsto \left[L \mid x \xleftarrow{\diamond} N\right]^{\mathcal{S}} +\!\!+ \left[M \mid x \xleftarrow{\diamond} N\right]^{\text{set}} \\
\left[L \mid x \xleftarrow{\diamond} M +\!\!+ N\right]^{\text{set}} \leadsto \left[L \mid x \xleftarrow{\diamond} M\right]^{\text{set}} +\!\!+ \left[L \mid x \xleftarrow{\diamond} N\right]^{\text{set}} \\
[L \mid x \leftarrow M +\!\!+ N]^{\text{bag}} \leadsto [L \mid x \leftarrow M]^{\text{bag}} +\!\!+ [L \mid x \leftarrow N]^{\text{bag}} \\
\left[L \mid k \xleftarrow{\mathcal{K}} M +\!\!+ N\right]^{\text{bag}} \not\leadsto \left[L \mid k \xleftarrow{\mathcal{K}} M\right]^{\text{bag}} +\!\!+ \left[L \mid k \xleftarrow{\mathcal{K}} N\right]^{\text{bag}}
\end{array}
$$

Finally, we have a look at the reduction rules involving grouping and lookup. They state that:

- empty collections are absorbent elements for both operations;
- grouping over a singleton maps ($\gamma_{x.\rho}([\rho' \triangleright M])$) extends the key ρ' of the singleton with new fields obtained from the grouping criterion ρ, instantiated on the value M;
- lookup on a singleton map ($[\rho \triangleright M] \circledast N$) reduces to a conditional statement evaluating to the singleton collection containing M if and only if $\langle\rho\rangle = N$, and to an empty collection otherwise;
- grouping and lookup commute with unions and disjoint unions, with the head of a comprehension, and with the left-hand argument of a **where**;

- nested γs can be merged by merging their grouping criteria;

Notice that there are no rewrite rules involving groupwise aggregation. Remember that the main purpose of our rewrite system is to simplify away unnecessarily nested intermediate collections: since the input and output types of *aggBy* are already flat, no rewrite rules are needed for our purposes (however, its argument M may still be involved in rewrites).

Remark. We can easily express the $\mathscr{G}$-comprehension from the introduction using key cmprehension and finite map application

$$\left[R \mid (k,v) \overset{\mathscr{G}}{\leftarrow} M\right] := \left[R\,[^{R \circledast k}/_{v}] \mid k \overset{\mathscr{K}}{\leftarrow} M\right]$$

The reason why we adopt $\overset{\mathscr{K}}{\leftarrow}$ as primitive instead of $\overset{\mathscr{G}}{\leftarrow}$ is that it has better algebraic properties. It can be shown that $\overset{\mathscr{G}}{\leftarrow}$ does not distribute over unions:

$$\left[R \mid (k,v) \overset{\mathscr{G}}{\leftarrow} M +\!\!\!+ N\right] \not\rightsquigarrow \left[R \mid (k,v) \overset{\mathscr{G}}{\leftarrow} M\right] +\!\!\!+ \left[R \mid (k,v) \overset{\mathscr{G}}{\leftarrow} N\right]$$

By unfolding the definition of $\overset{\mathscr{G}}{\leftarrow}$, we see that, in order for such a rule to be sound, we would need substitution to commute with unions, but that is generally not the case ($R\,[^{M+\!\!+N}/_{v}] \not\equiv R\,[^{M}/_{v}] +\!\!\!+ R\,[^{N}/_{v}]$).

3 Strong normalization

We now move to proving an important termination property of the rewrite system presented in the previous section: all well-typed $\mathcal{NRC}_{\lambda\gamma}$ terms will eventually be reduced to their normal form in a finite number of steps, regardless of the reduction strategy. This property is known as *strong normalization.*

Our proof is derived as a corollary of the strong normalization proof for the smaller calculus $\mathcal{NRC}_{\lambda}(Set, Bag)$, via a translation. We prove that every well-typed $\mathcal{NRC}_{\lambda\gamma}$ term can be translated to a well-typed $\mathcal{NRC}_{\lambda}(Set, Bag)$ term, and to every reduction step on well-typed $\mathcal{NRC}_{\lambda\gamma}$ terms there correspond one or more reduction steps on $\mathcal{NRC}_{\lambda}(Set, Bag)$ via the same translation procedure. Then, since there are no infinite reduction sequences in $\mathcal{NRC}_{\lambda}(Set, Bag)$, there can be no infinite reduction sequences in $\mathcal{NRC}_{\lambda\gamma}$.

The embedding of $\mathcal{NRC}_{\lambda\gamma}$ into $\mathcal{NRC}_{\lambda}(Set, Bag)$ is defined in Figure 5. The rule for grouping is type directed because we need to know the row associated to the grouping input (or in other words, the labels ℓ within the tuple $x.1$). A minor complication is the translation of aggregations (both simple and groupwise), since they have no counterpart in $\mathcal{NRC}_{\lambda}(Set, Bag)$; we map both of them to a distinguished unary constant $\bullet$ – this is sufficient because aggregations do not participate in reductions.

Lemma 3.1. *If $\Gamma \vdash M : T$ in $\mathcal{NRC}_{\lambda\gamma}$, then $\lceil \Gamma \rceil \vdash \lceil M \rceil : \lceil T \rceil$ in $\mathcal{NRC}_{\lambda}(Set, Bag)$.*

Theorem 3.2. *Whenever $M \rightsquigarrow N$ in $\mathcal{NRC}_{\lambda\gamma}$, we have that $\lceil M \rceil \overset{+}{\rightsquigarrow} \lceil N \rceil$ in $\mathcal{NRC}_{\lambda}(Set, Bag)$.*

The proof is by induction and case analysis on the reduction rule, unfolding the definition of the embedding as needed. Some care is needed when handling reduction rules involving $\gamma_{x.\rho}(()M)$, due to the more involved definition of the corresponding embedding. More details of the proof are given in the appendix.

Theorem 3.3. *If M is a well-typed $\mathcal{NRC}_{\lambda\gamma}$ term, there is no infinite reduction sequence starting with it.*

Proof. By Theorem 3.2, every infinite $\mathcal{NRC}_{\lambda\gamma}$ reduction sequence starting with M can be simulated by an infinite $\mathcal{NRC}_{\lambda}(Set, Bag)$ reduction sequence starting with $\lceil M \rceil$. By Lemma 3.1, since M is well-typed in $\mathcal{NRC}_{\lambda\gamma}$, $\lceil M \rceil$ is well-typed in $\mathcal{NRC}_{\lambda}(Set, Bag)$. But given that well-typed $\mathcal{NRC}_{\lambda}(Set, Bag)$ terms are strongly normalizing, there can be no infinite reduction starting with $\lceil M \rceil$. Consequently, M must be strongly normalizing in $\mathcal{NRC}_{\lambda\gamma}$. □

4 Grammar of normal forms

The remaining results of this paper rely on the structure of $\mathcal{NRC}_{\lambda\gamma}$ normal forms. To ease the reasoning on such terms, it will be useful to derive a grammar of normal forms, which can be used to perform case analysis and recursion without considering all of the cases of the general grammar of terms, many of which are precluded by the fact that the term has been normalized and must therefore be pruned.

We actually go a step further and add to normalization a few "administrative" rules and assumptions: in so doing, we effectively present "standardized" normal forms for which the following conditions are met:

- n-ary unions $C_1 +\!\!\!+^{\mathrm{set}} \cdots +\!\!\!+^{\mathrm{set}} C_n$ and $D_1 +\!\!\!+^{\mathrm{bag}} \cdots +\!\!\!+^{\mathrm{bag}} D_n$ are represented as $\bigcup \overrightarrow{C}$ and $\biguplus \overrightarrow{D}$ respectively; for empty $\overrightarrow{C}$ or $\overrightarrow{D}$, their grand unions stand for empty collections $[]^{\mathrm{set}}$, $[]^{\mathrm{bag}}$;
- variables with record type and comprehensions whose head is not a singleton are kept in eta-long form using the following rules:

$$\frac{x : \langle \overrightarrow{\ell : T} \rangle}{x \rightsquigarrow \langle \overrightarrow{\ell = x.\ell} \rangle}$$

$$\frac{M : [T]^{\mathcal{S}}_{\overrightarrow{\ell : S}} \qquad N = \left[\overrightarrow{\ell = k.\ell} \triangleright v\right]}{\left[M \textbf{ where } X \mid \overrightarrow{F}\right]^{\mathcal{S}} \rightsquigarrow \left[N \textbf{ where } X \mid \overrightarrow{F}, k \overset{\mathscr{K}}{\leftarrow} M, v \leftarrow M \circledast k\right]^{\mathcal{S}}}$$

- comprehensions without a guard are considered syntactically equal to those with a trivial guard:

$$\left[[\rho \triangleright M] \mid x \overset{\diamond}{\leftarrow} N\right] = \left[[\rho \triangleright M] \textbf{ where } \mathsf{true} \mid x \overset{\diamond}{\leftarrow} N\right]$$

$$\lceil b \rceil = b \qquad \lceil S \to T \rceil = \lceil S \rceil \to \lceil T \rceil \qquad \lceil \langle P \rangle \rceil = \langle \lceil P \rceil \rangle$$
$$\left\lceil [T]^{\mathcal{S}}_{P} \right\rceil = [\langle\langle \lceil P \rceil \rangle, \lceil T \rceil \rangle]^{\mathcal{S}} \qquad \left\lceil \overrightarrow{\ell : T} \right\rceil = \overrightarrow{\ell : \lceil T \rceil}$$

$$\left\lceil c(\overrightarrow{M}) \right\rceil = c(\overrightarrow{M}) \qquad \lceil \langle \rho \rangle \rceil = \langle \lceil \rho \rceil \rangle \qquad \lceil M.\ell \rceil = \lceil M \rceil.\ell \qquad \left\lceil \overrightarrow{\ell = M} \right\rceil = \overrightarrow{\ell = \lceil M \rceil}$$
$$\left\lceil \lambda x^{T}.M \right\rceil = \lambda x^{\lceil T \rceil}.\lceil M \rceil \qquad \lceil M\ N \rceil = \lceil M \rceil\ \lceil N \rceil \qquad \left\lceil M \ \mathbf{where}^{\mathcal{S}}\ N \right\rceil = \lceil M \rceil\ \mathbf{where}^{\mathcal{S}}\ \lceil N \rceil$$
$$\left\lceil []^{\mathcal{S}} \right\rceil = []^{\mathcal{S}} \qquad \left\lceil [\rho \triangleright M]^{\mathcal{S}} \right\rceil = [\langle\langle \lceil \rho \rceil \rangle, \lceil M \rceil \rangle]^{\mathcal{S}} \qquad \left\lceil M +\!\!\!+^{\mathcal{S}} N \right\rceil = \lceil M \rceil +\!\!\!+^{\mathcal{S}} \lceil N \rceil$$
$$\lceil \delta M \rceil = \delta \lceil M \rceil \qquad \lceil \iota M \rceil = \iota \lceil M \rceil \qquad \left\lceil [M \mid x \leftarrow N]^{\mathcal{S}} \right\rceil = [\lceil M \rceil [{}^{x.2}/_{x}] \mid x \leftarrow \lceil N \rceil]^{\mathcal{S}}$$
$$\left\lceil \left[M \mid k \stackrel{\mathcal{K}}{\leftarrow} N\right]^{\mathrm{set}} \right\rceil = [\lceil M \rceil \mid k \leftarrow dom(\lceil N \rceil)]^{\mathrm{set}} \qquad \left\lceil \left[M \mid k \stackrel{\mathcal{K}}{\leftarrow} N\right]^{\mathrm{bag}} \right\rceil = [\lceil M \rceil \mid k \leftarrow \iota dom(\lceil N \rceil)]^{\mathrm{bag}}$$
$$\left\lceil \gamma^{\mathcal{S}}_{x.\rho}(M) \right\rceil = \left[\left[\langle\langle \overrightarrow{\ell = x.1.\ell} \oplus \lceil \rho \rceil [{}^{x.2}/_{x}] \rangle, x.2 \rangle \right]^{\mathcal{S}} \mid x \leftarrow \lceil M \rceil \right]^{\mathcal{S}}$$
$$\left\lceil M \circledast^{\mathcal{S}} N \right\rceil = \left[[z.2]^{\mathcal{S}}\ \mathbf{where}^{\mathcal{S}}\ z.1 = \lceil N \rceil \mid z \leftarrow \lceil M \rceil \right]^{\mathcal{S}} \qquad (z \notin \mathrm{FV}(N))$$
$$\lceil \alpha(M) \rceil = \left\lceil aggBy^{\mathcal{S}}_{\overrightarrow{z.\ell_n = \alpha_n(z.\ell'_n)}}(M) \right\rceil = \bullet(\lceil M \rceil) \qquad dom(M) = \left[[z.1]^{\mathrm{set}} \mid z \leftarrow M \right]^{\mathrm{set}}$$

Figure 5. Embedding of $\mathcal{NRC}_{\lambda\gamma}$ into $\mathcal{NRC}_{\lambda}(Set, Bag)$.

- singletons that do not appear as the head of a comprehension are represented as trivial comprehensions

$$[\rho \triangleright M] = [[\rho \triangleright M] \mid\ ]$$

Similar rules have been considered in previous versions of $\mathcal{NRC}$. Once the above additional rules are considered, the normal forms of the resulting rewrite system are described by the grammar in Figure 6, as stated by the following result.

Theorem 4.1. *Every well-typed term M of $\mathcal{NRC}_{\lambda\gamma}$ in normal form generated by the grammar in Figure 6.*

The proof is by induction on the structure of M, noticing that the induction hypothesis states that the subterms of the M for which we are proving this theorem are generated by the grammar.

4.1 Normal forms of relational maps

For the purpose of generating SQL queries, we are not interested in all the terms of $\mathcal{NRC}_{\lambda\gamma}$ in their full generality, but we will instead focus on particular types of terms that we intend to translate to SQL: we call these terms nested and flat relational maps.

A nested relational map (**nrm**) is a term of finite map type whose keys are rows of basic type, and whose values are, inductively, tuples of nested relational maps:

$$\frac{\rho = \overrightarrow{\mathbb{k} : b} \qquad \overrightarrow{(T_i\ \mathbf{nrm})}}{\left[\langle \overrightarrow{\ell : T} \rangle \right]_{\rho}\ \mathbf{nrm}} \qquad \frac{\rho = \overrightarrow{\mathbb{k} : b}}{\left[\langle \overrightarrow{\ell : b'} \rangle \right]_{\rho}\ \mathbf{nrm}}$$

Under the additional condition that function typed terms should not appear as the argument of deduplication δ and promotion ι (as in $\mathcal{NRC}_{\lambda}(Set, Bag)$), we can easily see that all function terms are normalized away and all the variables bound by comprehensions (equivalently, all bound variables *tout court*, since there can be no lambda abstractions in the

$$\begin{array}{llcl}
\textbf{General NF} & M & ::= & B \mid U \mid W \mid Q \mid R \\
 & \rho & ::= & \overrightarrow{\ell = M} \\
\textbf{Indeterminate NF} & X & ::= & x \mid X.\ell \mid X\ M \\
\textbf{Base-typed NF} & B & ::= & X \mid c(\overrightarrow{B}) \mid \alpha(R) \\
\textbf{Tuple-typed NF} & U & ::= & X \mid \langle \rho \rangle \\
\textbf{Function-typed NF} & W & ::= & X \mid \lambda x.M \\
\textbf{Set-typed NF} & Q & ::= & []^{\mathrm{set}} \mid C \mid Q +\!\!\!+^{\mathrm{set}} Q \\
 & C & ::= & H \mid [C \mid F]^{\mathrm{set}} \\
 & H & ::= & I \mid I\ \mathbf{where}^{\mathrm{set}}\ B \mid Y \\
 & I & ::= & N \mid [\rho \triangleright M]^{\mathrm{set}} \\
 & N & ::= & X \mid \gamma^{\mathrm{set}}_{x.\rho}(N) \mid N \circledast^{\mathrm{set}} U \\
 & & \mid & \delta X \mid \delta t \mid Y \\
 & F & ::= & x \leftarrow N \mid k \stackrel{\mathcal{K}}{\leftarrow} N \\
 & Y & ::= & aggBy^{\mathrm{set}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}}(Q) \\
\textbf{Bag-typed NF} & R & ::= & []^{\mathrm{bag}} \mid D \mid R +\!\!\!+^{\mathrm{bag}} R \\
 & D & ::= & J \mid [D \mid G]^{\mathrm{bag}} \mid Z \\
 & J & ::= & L \mid L\ \mathbf{where}^{\mathrm{bag}}\ B \\
 & L & ::= & O \mid [\rho \triangleright M]^{\mathrm{bag}} \\
 & O & ::= & X \mid \gamma^{\mathrm{bag}}_{x.\rho}(O) \mid O \circledast^{\mathrm{bag}} U \\
 & & \mid & t \mid \iota Q \mid Z \\
 & G & ::= & x \leftarrow O \mid k \stackrel{\mathcal{K}}{\leftarrow} Q \\
 & Z & ::= & aggBy^{\mathrm{bag}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}}(R)
\end{array}$$

Figure 6. Normal forms of $\mathcal{NRC}_{\lambda\gamma}$

normal forms of relational maps) have a tuple type containing either base types or, inductively, nested relational maps.

Thanks to these considerations, we can derive a further simplified grammar of nested relational maps, described by Figure 7. This grammar is made slightly more compact by

$$
\begin{array}{rcl}
M & ::= & B \mid Q \mid R \\
\rho & ::= & \overrightarrow{\ell = M} \\
\rho^* & ::= & \overrightarrow{\ell = B} \\
B & ::= & x.\ell \mid c(\overrightarrow{B}) \mid \alpha(R) \\
Q & ::= & \bigcup \overrightarrow{C} \\
C & ::= & \left[[\rho^* \triangleright \langle \rho \rangle]^{\mathrm{set}} \ \mathbf{where}^{\mathrm{set}} \ B \mid \overrightarrow{F} \right] \mid Y \\
N & ::= & x.\ell \mid \gamma^{\mathrm{set}}_{x.\rho^*}(N) \mid N \circledast^{\mathrm{set}} \langle \rho^* \rangle \mid \delta(x.\ell) \mid \delta t \mid Y \\
Y & ::= & aggBy^{\mathrm{set}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}}(Q^*) \\
F & ::= & x \leftarrow N \mid k \xleftarrow{\mathcal{K}} N \\
R & ::= & \biguplus \overrightarrow{D} \\
D & ::= & \left[[\rho^* \triangleright \langle \rho \rangle]^{\mathrm{bag}} \ \mathbf{where}^{\mathrm{bag}} \ B \mid \overrightarrow{G} \right]^{\mathrm{bag}} \mid Z \\
O & ::= & x.\ell \mid \gamma^{\mathrm{set}}_{x.\rho^*}(O) \mid O \circledast^{\mathrm{bag}} \langle \rho^* \rangle \mid t \mid \iota Q^* \mid Z \\
G & ::= & x \leftarrow O \mid k \xleftarrow{\mathcal{K}} Q^* \\
Z & ::= & aggBy^{\mathrm{bag}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}}(R^*)
\end{array}
$$

(Q^* and R^* are defined similarly to Q and R, but replacing deep references to ρ (i.e. those in C and D) with ρ^*.)

Figure 7. Nested relational map normal forms of $\mathcal{NRC}_{\lambda\gamma}$

$$
\begin{array}{rcl}
M & ::= & B \mid Q^* \mid R^* \\
\rho^* & ::= & \overrightarrow{\ell = B} \\
B & ::= & x.\ell \mid c(\overrightarrow{B}) \mid \alpha(R^*) \\
Q^* & ::= & \bigcup \overrightarrow{C^*} \\
C^* & ::= & \left[[\rho^* \triangleright \langle \rho^* \rangle]^{\mathrm{set}} \ \mathbf{where}^{\mathrm{set}} \ B \mid \overrightarrow{F^*} \right] \mid Y \\
N^* & ::= & \gamma^{\mathrm{set}}_{x.\rho^*}(N^*) \mid N^* \circledast^{\mathrm{set}} \langle \rho^* \rangle \mid \delta t \mid Y \\
F^* & ::= & x \leftarrow N^* \mid k \xleftarrow{\mathcal{K}} N^* \\
Y & ::= & aggBy^{\mathrm{set}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}}(Q^*) \\
R^* & ::= & \biguplus \overrightarrow{D^*} \\
D^* & ::= & \left[[\rho^* \triangleright \langle \rho^* \rangle]^{\mathrm{bag}} \ \mathbf{where}^{\mathrm{bag}} \ B \mid \overrightarrow{G^*} \right]^{\mathrm{bag}} \mid Z \\
O^* & ::= & \gamma^{\mathrm{bag}}_{x.\rho^*}(O^*) \mid O^* \circledast^{\mathrm{bag}} \langle \rho^* \rangle \mid t \mid \iota Q^* \mid Z \\
G^* & ::= & x \leftarrow O^* \mid k \xleftarrow{\mathcal{K}} Q^* \\
Z & ::= & aggBy^{\mathrm{bag}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}}(R^*)
\end{array}
$$

Figure 8. Flat relational map normal forms of $\mathcal{NRC}_{\lambda\gamma}$

means of syntactic sugar for multi-generator comprehensions, representing as usual head-nested comprehensions.

A final simplification consists in limiting relational maps to the flat case, in which no collection nesting on the values of a map is allowed: in other words, flat relational maps are terms whose type is in the form $\left[\langle \overrightarrow{\ell : b'} \rangle\right]_{\overrightarrow{k:b}}$.

Applying this constraint to the grammar for nested relational normal forms we obtain the grammar in Figure 8.

5 Translation to SQL

The grammar of normal forms is amenable to be used to define algorithmic procedures, including a translation of $\mathcal{NRC}_{\lambda\gamma}$ normal forms to SQL; clearly, only flat normal forms need to be translated, as nested queries do not have a direct representation in SQL due to typing limitations. We devised one such translation and used it to implement language-integrated database queries with grouping and aggregation in the Links programming language [7]: due to space constraints, we give here a high level description of the translation of comprehension forms, referring the reader to the appendix for further details.

To translate comprehensions, our procedure operates by returning a complete `SELECT` … `FROM` … `WHERE` … statement: for instance

$$
\begin{aligned}
&\left(\left[[\rho \triangleright \langle \rho' \rangle]^{\mathrm{bag}} \ \mathbf{where}^{\mathrm{bag}} \ X \mid \overrightarrow{G} \right]^{\mathrm{bag}}\right)^{\mathrm{sql}} \\
&= \texttt{SELECT} \ (\rho)^{\mathrm{sql}}_K, (\rho')^{\mathrm{sql}}_V \ \texttt{FROM} \ \overrightarrow{(G)^{\mathrm{sql}}} \ \texttt{WHERE} \ (X)^{\mathrm{sql}}
\end{aligned}
$$

(with or without `DISTINCT`, depending on whether we are performing a set or a bag comprehension); the **where** clause is translated to the `WHERE` statement, and the comprehension generators are transated to the `FROM` clause; that leaves the singleton key-value pair – namely, $[\rho \triangleright \langle \rho' \rangle]$; both ρ and ρ' will go to the `SELECT` statement, but ρ shall use the auxiliary $(\rho)^{\mathrm{sql}}_K$ translation, marking that row as a key, while ρ' shall use the $(\rho')^{\mathrm{sql}}_V$ translation, marking it as a value row. Concretely, the two different translations rename the attributes using prefixes that univocally identify them as keys or values. Generators G are translated differently depending on whether they represent value comprehensions or key comprehensions: value comprehensions use a simple recursive call $(G)^{\mathrm{sql}}$; key comprehensions use a specialized auxiliary translation that removes the prefix from the attributes of $(G)^{\mathrm{sql}}$ that are marked as keys, and drops the attributes marked as values entirely.

5.1 Query shredding and tabular functions

Previous work [4] introduced *query shredding*, a technique which allows database queries with a nested collection type to be decomposed (i.e., "shredded") into multiple flat queries which can be expressed in SQL, thus run by a typical SQL-based DBMS, yielding partial results that are then *stitched* back together into the desired final value having a nested relational type. More recently, [22] proposed $\mathcal{NRC}_{\mathcal{G}}$, an extension of $\mathcal{NRC}$ with tabular functions which greatly simplifies the study of query shredding. $\mathcal{NRC}_{\mathcal{G}}$ shares some similarities with the finite maps resulting from the grouping operators of our $\mathcal{NRC}_{\lambda\gamma}$. In $\mathcal{NRC}_{\mathcal{G}}$, tabular functions are created using the graph operators $\mathcal{G}^{\mathrm{set}}(-;-)$ and $\mathcal{G}^{\mathrm{bag}}(-;-)$. We compare the graph operator with the grouping operator from $\mathcal{NRC}_{\lambda\gamma}$ in the set case (note that we have taken the liberty of adapting the types the subterms of $\mathcal{G}^{\text{set}}(-;-)$ to match them to the style used in this paper):

$$\frac{\Gamma \vdash L : \left[\overrightarrow{P}\right]^{\text{set}} \qquad \Gamma, \overrightarrow{x : P} \vdash M : [T]^{\text{set}}}{\Gamma \vdash \mathcal{G}^{\text{set}}(x \leftarrow L; M) : [T]^{\text{set}}_{P}} \qquad \frac{\Gamma \vdash M : [T]^{\text{set}} \qquad \Gamma, \overrightarrow{x : T} \vdash \rho : P}{\Gamma \vdash \gamma^{\text{set}}_{x.\rho}(M) : [T]^{\text{set}}_{P}}$$

On the right-hand side, the grouping constructor $\gamma^{\text{set}}_{x.\rho}(M)$ works by producing a different key ρ for each element x of the input collection M: this effectively groups the elements of M sharing the same key ρ; on the left-hand side, the graph constructor $\mathcal{G}^{\text{set}}(x \leftarrow L; M)$ takes the elements x of domain collection L and returns for each of them a codomain collection M which may depend on x: we may view this finite map as a "grouping", where the grouping keys are the elements of L.

The tabular functions of $\mathcal{NRC}_{\mathcal{G}}$ cannot be used in aggregations, but they do allow access to "groups" using the lookup syntax $\circledast$ (which we borrowed for $\mathcal{NRC}_{\lambda\gamma}$).

While the lookup operation plays a very similar role in the two calculi, the introduction rules for tabular functions/finite maps are rather dissimilar. Despite the superficial differences, we can show that each operation can be expressed in terms of the other:

$$\begin{array}{rcl} \mathcal{G}^{\text{set}}(x \leftarrow L; M) & ::= & \left[\gamma^{\text{set}}_{\_.\rho_x}(M) \mid x \leftarrow L\right] \\ \gamma^{\text{set}}_{x.\rho}(M) & ::= & \left[\mathcal{G}^{\text{set}}(\_ \leftarrow [\rho]^{\text{set}}; [x]^{\text{set}}) \mid x \leftarrow M\right] \end{array}$$

(where _ is any variable name chosen to be fresh with respect to its scope, and ρ_x is the row corresponding to the eta-expansion of the tuple-typed x, defined as $\overrightarrow{\ell = x.\ell}$, whenever x has type $\langle\overrightarrow{\ell : S}\rangle$; additionally, in the second equation above we have extended the $\mathcal{NRC}_{\mathcal{G}}$ notion of comprehension to allow it to return a tabular function, which was not allowed in the original formulation but is a straightforward extension nonethelesss).

Therefore, $\mathcal{NRC}_{\lambda\gamma}$ is very similar to $\mathcal{NRC}_{\mathcal{G}}$; however, $\mathcal{NRC}_{\mathcal{G}}$ did not provide any rewrite system and was largely intended as syntactic sugar that could be translated to plain $\mathcal{NRC}_{\lambda}$. On the contrary, $\mathcal{NRC}_{\lambda\gamma}$ provides its own notion of reduction and its normal forms can be directly used to implement language-integrated database queries.

5.2 A query shredding judgment for $\mathcal{NRC}_{\lambda\gamma}$

Since $\mathcal{G}^{\text{set}}(-;-)$ can be intended as a defined operator within $\mathcal{NRC}_{\lambda\gamma}$, we wonder whether it is possible to extend query shredding to $\mathcal{NRC}_{\lambda\gamma}$, thus allowing one to run nested relational queries involving grouping operations. The answer is affirmative and it involves, as it turns out, a limited redesign of the shredding judgment from [22], of which we show some key rules in Figure 9. As in the case of $\mathcal{NRC}_{\lambda}(Set, Bag)$, the shredding algorithm performs structural recursion over the grammar of nested relational normal forms; in $\mathcal{NRC}_{\lambda\gamma}$, however, we will need to consider some more cases for nested relational map normal forms that are not pure collections.

$$\frac{\varphi \notin dom(\Psi) \qquad \Phi; \Theta, \overrightarrow{F} \vdash \rho \Rrightarrow \breve{\rho} \mid \Psi}{\begin{array}{ll}\Phi; \Theta \vdash & \left[[\rho^* \triangleright \langle\rho\rangle]^{\text{set}} \textbf{ where } B \mid \overrightarrow{F}\right]^{\text{set}} \Rrightarrow \varphi \circledast^{\text{set}} \langle dom(\Theta)\rangle \\ \mid & \Psi[\varphi \mapsto \mathcal{G}(\Theta; \left[[\rho^* \triangleright \breve{\rho}]^{\text{set}} \textbf{ where } B \mid \overrightarrow{F}\right]^{\text{set}})]\end{array}}$$

$$\frac{\varphi \notin dom(\Psi) \qquad \Phi; \Theta, \overrightarrow{G^{\delta}} \vdash \rho \Rrightarrow \breve{\rho} \mid \Psi}{\begin{array}{ll}\Phi_0; \Theta \vdash & \left[[\rho^* \triangleright \rho]^{\text{bag}} \textbf{ where } B \mid \overrightarrow{G}\right]^{\text{bag}} \Rrightarrow \varphi \circledast^{\text{bag}} \langle dom(\Theta)\rangle \\ \mid & \Psi[\varphi \mapsto \mathcal{G}(\Theta; \left[[\rho^* \triangleright \breve{\rho}]^{\text{bag}} \textbf{ where } B \mid \overrightarrow{G}\right]^{\text{bag}})]\end{array}}$$

$$(x \leftarrow O)^{\delta} \triangleq \begin{cases} x \leftarrow Q^* & \text{if } O = \iota Q^* \\ x \leftarrow \delta O & \text{else} \end{cases} \qquad \Phi \setminus \overrightarrow{\psi} \triangleq [(\varphi \mapsto N) \in \Phi \mid \varphi \notin \overrightarrow{\psi}]$$

$$(k \overset{\mathcal{K}}{\leftarrow} Q^*)^{\delta} \triangleq Q^*$$

Figure 9. Shredding rules.

The *shredding judgment* describes the process by which, given a normalized $\mathcal{NRC}_{\lambda\gamma}$ query, each of its subqueries having a nested finite map type is lifted (in a manner analogous to lambda-lifting [12]) to an independent finite map query: more specifically, shredding will produce a *shredding environment* (denoted by $\Phi, \Psi, \ldots$), which is a finite map associating special *graph variables* φ, ψ to $\mathcal{NRC}_{\lambda\gamma}$ terms:

$$\Phi, \Psi, \ldots ::= [\overrightarrow{\varphi \mapsto M}]$$

The shredding judgment has the following form:

$$\Phi; \Theta \vdash M \Rrightarrow \breve{M} \mid \Psi$$

where the $\Rrightarrow$ symbol separates the input (to the left) from the output (to the right). The normalized $\mathcal{NRC}_{\lambda\gamma}$ term M is the query that is being considered for shredding; M may contain free variables declared in Θ, which must be a sequence of $\mathcal{NRC}_{\lambda}(Set, Bag)$ set comprehension bindings. Θ is initially empty, but during shredding it is extended with parts of the input that have already been processed. Similarly, the input shredding environment Φ is initially empty, but will grow during shredding to collect shredded queries that have already been generated.

The output of shredding consists of a shredded term $\breve{M}$ and an output shredding environment Ψ. Ψ extends Φ with the new queries obtained by shredding M; $\breve{M}$ is an output $\mathcal{NRC}_{\lambda\gamma}$ query obtained from M by lifting its collection typed subqueries to independent flat queries defined in Ψ.

The shredding of finite maps in normal form (i.e. unions, comprehensions, and groupwise aggregations) is performed by means of *query lifting*: we turn the collection into a globally defined (graph) query, which will be associated to a fresh name φ and instantiated to the local comprehension context by graph application. This operation converts local subterms into global graphs: thus, when shredding a map, besides processing its subterms recursively, we will need to extend the output shredding environment with a definition for the new global graph φ. In the interesting case of comprehensions, φ is defined by graph-abstracting over the comprehension context Θ; notice that, since we are only shredding normalized terms, we are allowed to reason on the limited number of cases allowed by the grammar of normal forms and, in particular, it is easy to see that the judgment for bag comprehensions must ensure that generators $\overrightarrow{G}$ be converted into sets using the operation G^{δ}.

6 Related work

This paper follows a line of research aat the intersection of programming languages and databases encompassing over three decades. Inspired by Trinder and Wadler's work on understanding database queries as a form of monadic comprehension syntax [31], Buneman et al. developed the nested relational calculus [1, 2, 29] with nested collections and first-order functions. Wong [32] proved the *conservativity theorem* stating that any query computable in that language can be expressed without resorting to subcomputations with a greater degree of nesting than that of its input or output (whichever is greater), and provided a strongly normalizing rewrite system to simplify queries involving subcomputations of an unnecessarily nested type; this also implied that flat-flat $\mathcal{NRC}$ queries could be translated to SQL, allowing a practical implementation [33].

By adding higher-order functions, Cooper extended $\mathcal{NRC}$ to a superset of the simply-typed lambda calculus [6]. Support for *heterogeneous* queries mixing collection kinds such as sets and bags, initially advocated by Grust and Scholl [10], was added by Ricciotti and Cheney who also gave a proof of strong normalization of their rewrite system [21, 24]. Advancements in the formal semantics of SQL by Guagliardo and Libkin [11] also made it possible to give a mechanized proof that $\mathcal{NRC}$ queries using sets and bags can be translated to SQL with `LATERAL` joins [23].

Further research developed language-integrated query, i.e. techniques to integrate a domain-specific database query sublanguage into a general-purpose host language. Microsoft gave a commercial implementation, termed LINQ [16, 28], and Cheney et al. showed how to use normalization techniques to improve the reliability and performance of LINQ queries [3]. Language-integrated query was also implemented, in a form closely based on $\mathcal{NRC}$, in the Links programming language [15], which uses an effect system to identify computations that can be run as database queries. Furthermore, language-integrated query is available in Scala and Haskell through libraries such as Quill [20] and DSH [9].

Particulaly relevant to our paper is the research devoted to translating nested relational queries to multiple flat SQL queries. A shredding technique was devised by Cheney et al. [4] to accomplish this goal in $\mathcal{NRC}$ with bag semantics; shredding has also proved useful to achieve greater parallelism in large-scale distributed query processing [26]. A rationalization of shredding by Ricciotti and Cheney called *query lifting* [22] using ideas from the well-understood concept of lambda-lifting makes it possible to process queries mixing sets and bags; the definition of query lifting employs finite maps (partially inspired by [8]), which are used as the basis of this paper.

Various other works have proposed query calculi or language-integrated query facilities offering grouping and aggregation, incuding Libkin and Wong's $\mathcal{BQL}$ [14], Suzuki et al.'s QueΛ [13, 27], and Okura and Kameyama's Quelg [18, 19]. Unlike $\mathcal{NRC}_{\lambda\gamma}$, none of these works supports nested collections or first-class grouping independent of aggregation, but Quelg provides optimization techniques to produce more efficient SQL queries, which we view as complementary to our proposal. To enable the optimization of hybrid database and linear algebra workload, Shaikha et al. developed *semi-ring dictionaries* [25], which are conceptually similar to our finite maps; their work even proposes a variant of $\mathcal{NRC}$ allowing independent grouping, but crucially does not provide a rewrite system or normalization.

7 Conclusion

Our calculus $\mathcal{NRC}_{\lambda\gamma}$ provides a formalism which can be used to developed a principled implementation of language-integrated query with grouping and aggregation. Unlike other proposals, we allow grouping to happen independently of aggregation: this allows queries to be expressed in a way that is more natural in a general-purpose programming language, while at the same time making it possible to convert such queries to idiomatic SQL queries. Commercial implementations such as Microsoft's LINQ are known to fail on grouping queries due to an imperfect understanding of their theory [17]: we believe that $\mathcal{NRC}_{\lambda\gamma}$ provides as a rational basis to reason on such queries and are using it in an extension the Links programming language.

At the moment, $\mathcal{NRC}_{\lambda\gamma}$ does not address the efficient execution of queries: its normal forms are considerably more involved than those of $\mathcal{NRC}_{\lambda}$or even $\mathcal{NRC}_{\lambda}(\mathit{Set}, \mathit{Bag})$, suggesting that there is probably space for optimizations, particularly in the case of complex queries obtained compositionally from simpler ones, as shown by Okura and Kameyama in the context of the Quelg language [19]: we do expect to be able to translate their optimizations to $\mathcal{NRC}_{\lambda\gamma}$, and the experimental evaluation of this (possibly together with more optimizations) will be the subject of our future work.

References

[1] Buneman, P., Libkin, L., Suciu, D., Tannen, V., Wong, L.: Comprehension syntax. SIGMOD Record **23** (1994)

[2] Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. Theor. Comput. Sci.

149(1) (1995). https://doi.org/10.1016/0304-3975(95)00024-Q

[3] Cheney, J., Lindley, S., Wadler, P.: A practical theory of language-integrated query. In: ICFP (2013). https://doi.org/10.1145/2500365.2500586

[4] Cheney, J., Lindley, S., Wadler, P.: Query shredding: efficient relational evaluation of queries over nested multisets. In: SIGMOD. pp. 1027–1038. ACM (2014). https://doi.org/10.1145/2588555.2612186

[5] Cooper, E.: Programming language features for web application development. Ph.D. thesis, University of Edinburgh (2009)

[6] Cooper, E.: The script-writer’s dream: How to write great SQL in your own language, and be sure it will succeed. In: DBPL (2009). https://doi.org/10.1007/978-3-642-03793-1_3

[7] Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: web programming without tiers. In: FMCO (2007). https://doi.org/10.1007/978-3-540-74792-5_12

[8] Gibbons, J., Henglein, F., Hinze, R., Wu, N.: Relational algebra by way of adjunctions. Proc. ACM Program. Lang. **2**(ICFP) (Jul 2018). https://doi.org/10.1145/3236781

[9] Giorgidze, G., Grust, T., Schreiber, T., Weijers, J.: Haskell boards the Ferry - database-supported program execution for Haskell. In: IFL. pp. 1–18. No. 6647 in LNCS, Springer-Verlag (2010)

[10] Grust, T., Scholl, M.H.: How to comprehend queries functionally. J. Intell. Inf. Syst. **12**(2-3), 191–218 (1999). https://doi.org/10.1023/A:1008705026446

[11] Guagliardo, P., Libkin, L.: A formal semantics of SQL queries, its validation, and applications. PVLDB (2017). https://doi.org/10.14778/3151113.3151116

[12] Johnsson, T.: Lambda lifting: Treansforming programs to recursive equations. In: FPCA. pp. 190–203 (1985). https://doi.org/10.1007/3-540-15975-4_37

[13] Kiselyov, O., Katsushima, T.: Sound and efficient language-integrated query - maintaining the ORDER. In: APLAS 2017. pp. 364–383 (2017). https://doi.org/10.1007/978-3-319-71237-6_18

[14] Libkin, L., Wong, L.: Query languages for bags and aggregate functions. J. Comput. Syst. Sci. **55**(2) (1997)

[15] Lindley, S., Cheney, J.: Row-based effect types for database integration. In: TLDI (2012). https://doi.org/10.1145/2103786.2103798

[16] Meijer, E., Beckman, B., Bierman, G.M.: LINQ: reconciling object, relations and XML in the .NET framework. In: SIGMOD (2006). https://doi.org/10.1145/1142473.1142552

[17] .NET EF Core: Issue #30173: Improve GroupBy support, https://github.com/dotnet/efcore/issues/30173

[18] Okura, R., Kameyama, Y.: Language-integrated query with nested data structures and grouping. In: FLOPS. pp. 139–158 (2020). https://doi.org/10.1007/978-3-030-59025-3_9

[19] Okura, R., Kameyama, Y.: Reorganizing queries with grouping. In: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. p. 50–62. GPCE 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3425898.3426960, https://doi.org/10.1145/3425898.3426960

[20] Quill: Compile-time language integrated queries for Scala. Open source project, https://github.com/getquill/quill

[21] Ricciotti, W., Cheney, J.: Mixing set and bag semantics. In: DBPL. pp. 70–73 (2019). https://doi.org/10.1145/3315507.3330202

[22] Ricciotti, W., Cheney, J.: Query lifting: Language-integrated query for heterogeneous nested collections. In: ESOP. pp. 579–606 (2021)

[23] Ricciotti, W., Cheney, J.: A formalization of sql with nulls. J. Autom. Reaasoning **66**, 989–1030 (2022). https://doi.org/10.1007/s10817-022-09632-4, https://doi.org/10.1007/s10817-022-09632-4

[24] Ricciotti, W., Cheney, J.: Strongly-normalizing higher-order relational queries. LMCS **18**(3) (2022)

[25] Shaikhha, A., Huot, M., Smith, J., Olteanu, D.: Functional collection programming with semi-ring dictionaries. Proc. ACM Program. Lang. **6**(OOPSLA1) (apr 2022). https://doi.org/10.1145/3527333, https://doi.org/10.1145/3527333

[26] Smith, J., Benedikt, M., Nikolic, M., Shaikhha, A.: Scalable querying of nested data. Proc. VLDB Endow. **14**(3), 445–457 (nov 2020). https://doi.org/10.14778/3430915.3430933, https://doi.org/10.14778/3430915.3430933

[27] Suzuki, K., Kiselyov, O., Kameyama, Y.: Finally, safely-extensible and efficient language-integrated query. In: PEPM. pp. 37–48 (2016). https://doi.org/10.1145/2847538.2847542

[28] Syme, D.: Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In: ML Workshop (2006)

[29] Tannen, V., Buneman, P., Wong, L.: Naturally embedded query languages. In: ICDT. LNCS, vol. 646. Springer (1992)

[30] Transaction processing performance council: TPC-H benchmark (2022), http://www.tpc.org

[31] Trinder, P., Wadler, P.: Improving list comprehension database queries. In: TENCON ’89. (1989). https://doi.org/10.1109/TENCON.1989.176921

[32] Wong, L.: Normal forms and conservative extension properties for query languages over collection types. J. Comput. Syst. Sci. **52**(3) (1996). https://doi.org/10.1006/jcss.1996.0037

[33] Wong, L.: Kleisli, a functional query system. J. Funct. Programming **10**(1) (2000). https://doi.org/10.1017/S0956796899003585

A Proofs from Section 3

Lemma A.1. *For all M, N, and x, we have* $\lceil M\,[N/x]\rceil = \lceil M\rceil\,[\lceil N\rceil/x]$.

Proof. By structural induction on M: we comment on a few relevant cases:

- $\lceil(\lambda y.M)\,[N/x]\rceil = \lceil \lambda y.M\rceil\,[\lceil N\rceil/x]$: we assume without loss of generality that y is fresh with respect to x and N; by definition, the lhs is equal to $\lambda y.\,\lceil M\,[N/x]\rceil$, which can be rewritten to $\lambda y.\,\lceil M\rceil\,[\lceil N\rceil/x]$ by the IH; this is equal to the rhs by definition.
- $\left\lceil \left[L \mid k \overset{\mathcal{K}}{\leftarrow} M\right]^{\mathrm{set}}[N/x]\right\rceil = \left\lceil \left[L \mid k \overset{\mathcal{K}}{\leftarrow} M\right]^{\mathrm{set}}\right\rceil[\lceil N\rceil/x]$: we assume without loss of generality that k is fresh with respect to x and N; by definition, the lhs is equal to

$$[\lceil L\,[N/x]\rceil \mid k \leftarrow dom(\lceil M\,[N/x]\rceil)]^{\mathrm{set}}$$

  By the IH, we rewrite $\lceil L\,[N/x]\rceil$ to $\lceil L\rceil\,[\lceil N\rceil/x]$ and $\lceil M\,[N/x]\rceil$ to $\lceil M\rceil\,[\lceil N\rceil/x]$; then we can see that the resulting term is equal to the rhs by unfolding the definitions.
- $\left\lceil \gamma_{y.\rho}(M)\,[N/x]\right\rceil = \left\lceil \gamma_{y.\rho}(M)\right\rceil\,[\lceil N\rceil/x]$; we assume without loss of generality that y is fresh with respect to x and N; by definition, the lhs is equal to

$$\left[\left[\langle\langle\overrightarrow{\ell = y.1.\ell} \oplus \lceil \rho\,[N/x]\rceil\,[y.2/y]\rangle, y.2\rangle\right] \mid y \leftarrow \lceil M\,[N/x]\rceil\right]$$

  By the IH, we rewrite $\lceil M\,[N/x]\rceil$ to $\lceil M\rceil\,[\lceil N\rceil/x]$, and $\lceil \rho\,[N/x]\rceil$ to $\lceil\rho\rceil\,[\lceil N\rceil/x]$; by the substitution lemma, we prove $\lceil\rho\rceil\,[\lceil N\rceil/x]\,[y.2/y] = \lceil\rho\rceil\,[y.2/y]\,[\lceil N\rceil/x]$; by this, we can prove that the term is equal to the rhs.
- $\lceil(L \circledast M)\,[N/x]\rceil = \lceil L \circledast M\rceil\,[\lceil N\rceil/x]$; we choose z fresh with respect to x, L, M, N; then by definition, the lhs is equal to $[[z.2]$ **where** $z.1 = \lceil M\,[N/x]\rceil \mid z \leftarrow \lceil L\,[N/x]\rceil]$; by the IH, this is equal to

$$[[z.2]\ \textbf{where}\ z.1 = \lceil M\rceil\,[\lceil N\rceil/x] \mid z \leftarrow \lceil L\rceil\,[N/x]]$$

  which by definition is equal to the rhs. □

Theorem 3.2. Whenever $M \rightsquigarrow N$ in $\mathcal{NRC}_{\lambda\gamma}$, we have that $\lceil M\rceil \overset{+}{\rightsquigarrow} \lceil N\rceil$ in $\mathcal{NRC}_{\lambda}(Set, Bag)$.

Proof. By induction and case analysis on the reduction rule. We consider here some key cases:

- $\lceil(\lambda x.M)\,N\rceil \overset{+}{\rightsquigarrow} \lceil M\,[N/x]\rceil$: we prove that $\lceil M\,[N/x]\rceil = \lceil M\rceil\,[\lceil N\rceil/x]$. By Lemma A.1.
- $\left\lceil\gamma_{x.\rho}([])\right\rceil$
  $= \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho\rceil\,[x.2/x]\rangle, x.2\rangle\right] \mid x \leftarrow []\right]$
  $\overset{+}{\rightsquigarrow} [] = \lceil[]\rceil$: trivial.
- $\left\lceil\gamma_{x.\rho}([\rho' \triangleright M])\right\rceil$
  $= \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho\rceil\,[x.2/x]\rangle, x.2\rangle\right] \mid x \leftarrow [\langle\langle\lceil\rho'\rceil\rangle, \lceil M\rceil\rangle]\right]$
  $\overset{+}{\rightsquigarrow} \left[\langle\langle\overrightarrow{\ell = \lceil\rho'_\ell\rceil} \oplus \lceil\rho\rceil\,[\lceil M\rceil/x]\rangle, \lceil M\rceil\rangle\right]$
  $= \lceil[\rho\,[M/x] \oplus \rho' \triangleright M]\rceil$: we reduce the comprehension in the lhs, and then perform as many reductions on the resulting projection redexes as needed to obtain the rhs. This also uses Lemma A.1 just like the lambda-application case.
- $\left\lceil\gamma_{x.\rho}(\gamma_{x.\rho'}(M))\right\rceil =$
  $\left[\left[\langle\langle\overrightarrow{\ell = y.1.\ell} \oplus \lceil\rho\rceil\,[y.2/y]\rangle, y.2\rangle\right] \mid y \leftarrow \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho'\rceil\,[x.2/x]\rangle, x.2\rangle\right] \mid x \leftarrow \lceil M\rceil\right]\right]$
  $\overset{+}{\rightsquigarrow} \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho\rceil\,[x.2/x] \oplus \lceil\rho'\rceil\,[x.2/x]\rangle, x.2\rangle\right] \mid x \leftarrow \lceil M\rceil\right]$
  $= \left\lceil\gamma_{x.\rho\oplus\rho'}(M)\right\rceil$: we perform unnesting on the lhs, then reduce the comprehension binding y to a singleton; finally, we reduce the projections as many times as needed to obtain the rhs.
- $\left\lceil\gamma_{x.\rho}(M +\!\!\!+ N)\right\rceil$
  $= \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho\rceil\,[x/x.2]\rangle, x.2\rangle\right] \mid x \leftarrow \lceil M\rceil +\!\!\!+ \lceil N\rceil\right]$
  $\overset{+}{\rightsquigarrow} \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho\rceil\,[x/x.2]\rangle, x.2\rangle\right] \mid x \leftarrow \lceil M\rceil\right] +\!\!\!+ \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho\rceil\,[x/x.2]\rangle, x.2\rangle\right]^{\mathrm{set}} \mid x \leftarrow \lceil N\rceil\right]$
  $= \left\lceil\gamma_{x.\rho}(M) +\!\!\!+ \gamma_{x.\rho}(N)\right\rceil$: trivial.
- $\left\lceil\gamma_{x.\rho}([L \mid y \leftarrow M])\right\rceil$
  $= \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho\rceil\,[x/x.2]\rangle, x.2\rangle\right] \mid x \leftarrow [\lceil L\rceil\,[y.2/y] \mid y \leftarrow \lceil M\rceil]\right]$
  $\overset{+}{\rightsquigarrow} \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho\rceil\,[x.2/x]\rangle, x.2\rangle\right]\,[y.2/y] \mid y \leftarrow \lceil M\rceil, x \leftarrow \lceil L\rceil\,[y.2/y]\right]$
  $= \left\lceil\left[\gamma_{x.\rho}(L) \mid y \leftarrow M\right]\right\rceil$: we assume, withut loss of generality, that $y \neq x$ and $y \notin \mathrm{FV}(M, \rho)$; we perform unnesting on the lhs and, thanks to the freshness conditions, that the resulting term is equal to the rhs.
- $\left\lceil\gamma_{x.\rho}(\left[L \mid k \overset{\mathcal{K}}{\leftarrow} M\right]^{\mathrm{set}})\right\rceil$
  $= \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho\rceil\,[x/x.2]\rangle, x.2\rangle\right]^{\mathrm{set}} \mid x \leftarrow [\lceil L\rceil\,[k.1/k] \mid k \leftarrow \lceil M\rceil]^{\mathrm{set}}\right]^{\mathrm{set}}$
  $\overset{+}{\rightsquigarrow} \left[\left[\langle\langle\overrightarrow{\ell = x.1.\ell} \oplus \lceil\rho\rceil\,[x.2/x]\rangle, x.2\rangle\right]^{\mathrm{set}}\,[k.1/k] \mid y \leftarrow \lceil M\rceil, x \leftarrow \lceil L\rceil\,[y.1/k]\right]^{\mathrm{set}}$

$= \left\lceil \left[\gamma_{x.\rho}(L) \mid k \overset{\mathcal{K}}{\leftarrow} M \right] \right\rceil$: we assume, withut loss of generality, that $k \neq x$ and $k \notin \mathrm{FV}(M, \rho)$; we perform unnesting on the lhs and, thanks to the freshness conditions, that the resulting term is equal to the rhs.

- $\lceil [] \circledast N \rceil = [[\langle\langle\rangle, x.2\rangle] \ \mathbf{where}\ x.1 = \lceil N \rceil \mid x \leftarrow []]$
  $\overset{+}{\rightsquigarrow} [] = \lceil [] \rceil$: trivial.
- $\lceil (L \mathbin{+\!\!+} M) \circledast N \rceil$
  $= [[\langle\langle\rangle, x.2\rangle] \ \mathbf{where}\ x.1 = \lceil N \rceil \mid x \leftarrow \lceil L \rceil \mathbin{+\!\!+} \lceil M \rceil]$
  $\overset{+}{\rightsquigarrow} [[\langle\langle\rangle, x.2\rangle] \ \mathbf{where}\ x.1 = \lceil N \rceil \mid x \leftarrow \lceil L \rceil] \mathbin{+\!\!+}$
  $[[\langle\langle\rangle, x.2\rangle] \ \mathbf{where}\ x.1 = \lceil N \rceil \mid x \leftarrow \lceil M \rceil]$
  $= \lceil (L \circledast N) \mathbin{+\!\!+} (M \circledast N) \rceil$: trivial.
- $\lceil [L \mid x \leftarrow M] \circledast N \rceil$
  $$= \left[\begin{array}{l} [\langle\langle\rangle, y.2\rangle] \ \mathbf{where}\ y.1 = \lceil N \rceil \\ \quad \mid y \leftarrow [\lceil L \rceil\, [x.2/x] \mid x \leftarrow \lceil M \rceil] \end{array} \right]$$
  $$\overset{+}{\rightsquigarrow} \left[\begin{array}{l} [\langle\langle\rangle, y.2\rangle] \ \mathbf{where}\ y.1 = \lceil N \rceil \\ \quad \mid x \leftarrow \lceil M \rceil, y \leftarrow \lceil L \rceil\, [x.2/x] \end{array} \right]$$
  $= \lceil [L \circledast N \mid x \leftarrow M] \rceil$: we assume, without loss of generality, that $y \neq x$, and $y \notin \mathrm{FV}(N)$; we perform unnesting on the lhs and, by the freshness conditions, show that the resulting term is equal to the rhs.
- $\left\lceil \left[L \mid k \overset{\mathcal{K}}{\leftarrow} M \right]^{\mathrm{set}} \circledast N \right\rceil$
  $$= \left[\begin{array}{l} [\langle\langle\rangle, y.2\rangle] \ \mathbf{where}\ y.1 = \lceil N \rceil \\ \quad \mid y \leftarrow [\lceil L \rceil\, [k.1/k] \mid k \leftarrow \lceil M \rceil]^{\mathrm{set}} \end{array} \right]^{\mathrm{set}}$$
  $$\overset{+}{\rightsquigarrow} \left[\begin{array}{l} [\langle\langle\rangle, y.2\rangle] \ \mathbf{where}\ y.1 = \lceil N \rceil \\ \quad \mid k \leftarrow \lceil M \rceil, y \leftarrow \lceil L \rceil\, [k.1/k] \end{array} \right]^{\mathrm{set}}$$
  $= \left\lceil \left[L \circledast N \mid k \overset{\mathcal{K}}{\leftarrow} M \right]^{\mathrm{set}} \right\rceil$: we assume, without loss of generality, that $y \neq k$, and $y \notin \mathrm{FV}(N)$; we perform unnesting on the lhs and, by the freshness conditions, show that the resulting term is equal to the rhs.
- For all reductions happening within a context, the thesis is obtained by an application of the induction hypothesis. □

B Proofs from Section 4

Theorem 4.1. Every well-typed term M of $\mathcal{NRC}_{\lambda\gamma}$ in normal form generated by the grammar in Figure 6.

Proof. By induction on the structure of M. Notice that the induction hypothesis states that the subterms of the M for which we are proving this theorem are generated by the grammar. By cases:

- Terms in the form x, t, $\langle\rho\rangle$, $\lambda x.M$, $[]^{\mathcal{S}}$, $[\rho \triangleright M']^{\mathcal{S}}$: the proof is trivial, or a direct consequence of the induction hypothesis.
- $c(\overrightarrow{M'})$: by IH, $\overrightarrow{M'}$ is generated by the grammar; we reason by cases on the possible productions of $\overrightarrow{M'}$ and see that, in order for the term to be well-typed, we must have $\overrightarrow{M'} = \overrightarrow{B}$: $c(\overrightarrow{B})$ is generated by the grammar for M, which proves the thesis.
- $\alpha(M')$: by IH, M' is generated by the grammar; we reason by cases on the possible productions of M' and see that, in order for the term to be well-typed, we must have $M' = R$: $\alpha(R)$ is generated by the grammar for M, which proves the thesis.
- $M'.\ell$: by IH, M' is generated by the grammar; we reason by cases on the possible productions of M' and see that, in order for the term to be well-typed, we must have $M' = U$; U can be either X or $\langle\rho\rangle$, however $\langle\rho\rangle.\ell$ is not in normal form; that only leaves $X.\ell$, which is generated by the grammar for M, as required.
- $M'\ M''$: by IH, M' and M'' are generated by the grammar; we reason by cases on the possible productions of M' and see that, in order for the term to be well-typed, we must have $M' = W$; W can be either X or $\lambda x.M'''$, however $(\lambda x.M''')\ M''$ is not in normal form; that only leaves $X\ M''$, which is generated by the grammar for M, as required.
- $M' \mathbin{+\!\!+}^{\mathrm{set}} M''$: by IH, M' and M'' are generated by the grammar; we reason by cases on the possible productions of M' and M'' and see that, in order for the term to be well-typed, we must have $M' = Q'$, $M'' = Q''$; $Q' \mathbin{+\!\!+}^{\mathrm{set}} Q''$ is generated by the grammar for M, as required.
- M' $\mathbf{where}^{\mathrm{set}}$ M'': by IH, M' and M'' are generated by the grammar; we reason by cases on the possible productions of M' and M'' and see that, in order for the term to be well-typed, we must have $M' = Q$ and $M'' = B$; furthermore, in order for the term to be in normal form, we prove by deep case analysis on the possible productions of Q that we must have $Q = I$; then, I $\mathbf{where}^{\mathrm{set}}$ B is generated by the grammar for M, as required.
- $\delta M'$: by IH, M' is generated by the grammar; we reason by cases on the possible productions of M' and see that, in order for the term to be well-typed, we must have $M' = R$; by deep cases analysis on the productions starting at R, we see that, in order for δR to be in normal form, we must have $R = X$ or $R = t$; both δX and δt are generated by the grammar for M, as required.
- $\iota M'$: by IH, M' is generated by the grammar; we reason by cases on the possible productions of M' and see that, in order for the term to be well-typed, we must have $M' = Q$; ιQ is generated by the grammar for M, as required.
- $[M' \mid x \leftarrow M'']^{\mathrm{set}}$: by IH, M' and M'' are generated by the grammar; we reason by cases on the possible productions of M' and M'' and see that, in order for the term to be well-typed, we must have $M' = Q'$,

$M'' = Q''$; by more case analysis, we see that $Q' = C$ (otherwise, the term would not be in normal form); by a similar reasoning, Q'' must be in the form N: we thus easily show that $[C \mid x \leftarrow N]^{\mathsf{set}}$ is generated by the grammar for M, as required.

- $\left[M' \mid k \xleftarrow{\mathcal{K}} M''\right]^{\mathsf{set}}$: similar to the value comprehension case above.
- $\gamma^{\mathsf{set}}_{x.\rho}(M')$: by IH, M' is generated by the grammar; we reason by cases on the possible productions of M' and see that, in order for the term to be well-typed and in normal form, we must have $M' = N$; $\gamma^{\mathsf{set}}_{x.\rho}(N)$ is generated by the grammar for M, as required.
- $M' \circledast^{\mathsf{set}} M''$: by IH, M' and M'' are generated by the grammar; we reason by cases on the possible productions of M' and M'' and see that, in order for the term to be well-typed, we must have $M' = Q$, $M'' = U$; furthermore, by cases on the productions for Q, in order for the term to be in normal form we will need $Q = N$; then we see that the term $N \circledast^{\mathsf{set}} U$ is generated by the grammar, as required.
- $aggBy^{\mathsf{set}}_{\overrightarrow{\ell=\alpha}}(M')$: by IH, M' is generated by the grammar; we reason by cases on the possible productions of M' and see that, in order for the term to be well-typed, we must havee $M' = Q$; then we see that $aggBy^{\mathsf{set}}_{\overrightarrow{\ell=\alpha}}(Q)$ is generated by the grammar, as required.
- $M' +\!\!+^{\mathsf{bag}} M''$: by IH, M' and M'' are generated by the grammar; we reason by cases on the possible productions of M' and M'' and see that, in order for the term to be well-typed, we must have $M' = R'$, $M'' = R''$; $R' +\!\!+^{\mathsf{bag}} R''$ is generated by the grammar for M, as required.
- $[M' \mid x \leftarrow M'']^{\mathsf{bag}}$: by IH, M' and M'' are generated by the grammar; we reason by cases on the possible productions of M' and M'' and see that, in order for the term to be well-typed, we must have $M' = R'$, $M'' = R''$; by more case analysis, we see that $R' = D$ (otherwise, the term would not be in normal form); by a similar reasoning, R'' must be in the form O: we thus easily show that $[D \mid x \leftarrow O]^{\mathsf{bag}}$ is generated by the grammar for M, as required.
- $\left[M' \mid k \xleftarrow{\mathcal{K}} M''\right]^{\mathsf{bag}}$: by IH, M' and M'' are generated by the grammar; we reason by cases on the possible productions of M' and M'' and see that, in order for the term to be well-typed, we must have $M' = R'$, $M'' = Q''$; by more case analysis, we see that $R' = D$ (otherwise, the term would not be in normal form); we thus easily show that $\left[D \mid k \xleftarrow{\mathcal{K}} Q''\right]^{\mathsf{bag}}$ is generated by the grammar for M, as required.
- M' **where**$^{\mathsf{bag}}$ M'': by IH, M' and M'' are generated by the grammar; we reason by cases on the possible productions of M' and M'' and see that, in order for the term to be well-typed, we must have $M' = R$ and $M'' = B$; furthermore, in order for the term to be in normal form, we prove by deep case analysis on the possible productions of R that we must have $R = L$; then, L **where**$^{\mathsf{bag}}$ B is generated by the grammar for M, as required.
- $\gamma^{\mathsf{bag}}_{x.\rho}(M')$: by IH, M' is generated by the grammar; we reason by cases on the possible productions of M' and see that, in order for the term to be well-typed and in normal form, we must have $M' = O$; $\gamma^{\mathsf{bag}}_{x.\rho}(O)$ is generated by the grammar for M, as required.
- $M' \circledast^{\mathsf{bag}} M''$: by IH, M' and M'' are generated by the grammar; we reason by cases on the possible productions of M' and M'' and see that, in order for the term to be well-typed, we must have $M' = R$, $M'' = U$; furthermore, by cases on the productions for R, in order for the term to be in normal form we will need $R = O$; then we see that the term $O \circledast^{\mathsf{bag}} U$ is generated by the grammar, as required.
- $aggBy^{\mathsf{bag}}_{\overrightarrow{\ell=\alpha}}(M')$: by IH, M' is generated by the grammar; we reason by cases on the possible productions of M' and see that, in order for the term to be well-typed, we must havee $M' = R$; then we see that $aggBy^{\mathsf{bag}}_{\overrightarrow{\ell=\alpha}}(R)$ is generated by the grammar, as required. □

C Details of the translation to SQL (Section 5)

The grammar of normal forms is amenable to be used to define algorithmic procedures, including a translation of $\mathcal{NRC}_{\lambda\gamma}$ normal forms to SQL; clearly, only flat normal forms need to be translated, as nested queries do not have a direct representation in SQL due to typing limitations. We give one such translation in Figure 10: we define a main translation operation $(\cdot)^{\mathsf{sql}}$ along with auxiliary translations $(\cdot)^{\mathsf{sql}}_{K}$, $(\cdot)^{\mathsf{sql}}_{V}$, $(\cdot)^{\mathsf{sql}}_{GK}$, and operations *keys*, *vals*, *attrs*.

The main translation $(\cdot)^{\mathsf{sql}}$ assigns trivially empty queries to [] and, by recursion, union/disjoint union queries to $\mathcal{NRC}_{\lambda\gamma}$ queries employing $+\!\!+$. In the more involved comprehensions cases, the translation operates by returning a complete `SELECT` ... `FROM` ... `WHERE` ... statement (with or without `DISTINCT`, depending on whether we are performing a set or a bag comprehension); the **where** clause is translated to the `WHERE` statement, and the comprehension generators are transated to the `FROM` clause; that leaves the singleton key-value pair – namely, $[\rho \triangleright \langle\rho'\rangle]$; both ρ and ρ' will go to the `SELECT` statement, but ρ shall use the auxiliary $(\rho)^{\mathsf{sql}}_{K}$ translation, marking that row as a key, while ρ' shall use the $(\rho')^{\mathsf{sql}}_{V}$ translation, marking it as a value row. Concretely, the two different translations tag the attribute names so that attributes used as grouping keys (in the form

$1@\ell$) will always be distinguished from attributes used as values (in the form $2@\ell$).

The rest of the $\mathcal{NRC}_{\lambda\gamma}$ normal forms can only appear as comprehension generators: these include table references t (translated using `SELECT` with the value-tagging translation $(\cdot)^{\mathrm{sql}}_V$), deduplicated tables (which use `SELECT DISTINCT` $*$, promoted set queries (translated by recursion: from an SQL point of view, promotion does not change the semantics of a query).

The translation of grouping ($\gamma^{\mathrm{set}}_{x.\rho}(N)$ or $\gamma^{\mathrm{bag}}_{x.\rho}(O)$) uses a `SELECT` (`DISTINCT`) statement, in which the values from the table resulting from the evaluation of N (or O) are linked with keys $(\rho)^{\mathrm{sql}}_K$; notice that, although we know by reasoning on typing that N (or O) must be a pure collection (rather than a finite map), the recursive translation will return a table with value-tagged attributes: the $vals(\cdot)$ operation returns the list of value-tagged attributes. Dually, the lookup operation ($N \circledast^{\mathrm{set}} \langle\rho\rangle$ or $O \circledast^{\mathrm{bag}} \langle\rho\rangle$) also produces a `SELECT` (`DISTINCT`) statement uing the $vals(\cdot)$ operation, but the input relation N or O must be a proper finite map, thus we need to filter those keys matching the lookup row ρ. Groupwise aggregation is translated by means of a `SELECT` statement using the appropriate aggregates, along with a `GROUP BY` clause (notice that since we know that each key will be mapped to a singleton, we can avoid using `DISTINCT` even in the set case).

The main translation is used in this form to process value comprehensions; for key comprehensions, we provide a specific translation $(\cdot)^{\mathrm{sql}}_{GK}$ whose only cases are grouping, with set or bag semantics, and (implicitly promoted) set subqueries Q; the main purpose of this particular translation is to collect only the key attributes from the input map (using $keys(\cdot)$ in the inner set query case) and then retag them as value attributes; since we never want duplicate keys, not even in bags, all of the cases are translated using `DISTINCT`. In both kinds of comprehension, our translation can generate lateral joins (as it is normal for queries mixing sets an bags [21]) which require the SQL:1999 keyword `LATERAL`. Lateral joins can always be removed using the technique shown in [22] (in exchange for additional query complexity).

Aggregation can only be performed on pure collections of basic values: we convert these to collections of trivial unary tuples, translate them recursively, and finally apply the corresponding SQL aggregation operation to obtain the result we want.

As a concluding remark, we note that this translation procedure handles the grouping operator *without* employing SQL `GROUP BY` statements, save when it is the argument of a groupwise aggregation: this is because `GROUP BY` clauses can only be used when attributes other than the grouping keys have been aggregated.

Example C.1. The $\mathcal{NRC}_{\lambda\gamma}$ query M_{group} from Example 2.1 is translated to SQL as follows;

$(M_{group})^{\mathrm{sql}} :=$
```
SELECT x.1@dept AS 1@dept,
       AVERAGE(x.2@salary) as 2@salary
FROM (SELECT d.name AS 1@dept,
             e.salary AS 2@salary
      FROM department d, employee e
      WHERE d.id = e.dept)
GROUP BY 1@dept
```

This is slightly more complicated than the original query `q_group` from the introduction due to the separation of grouping and groupwise aggregation in the $\mathcal{NRC}_{\lambda\gamma}$ term; however, this kind of nesting is easily optimized in most DBMS; alternatively, we can get rid of such artifacts in a postprocessing step.

D Full definition of query shredding (Section 5.2)

Figure 11 shows the full definition of the shredding judgment for $\mathcal{NRC}_{\lambda\gamma}$. We comment the rules that were omitted from the main body of this article.

The rules for the shredding judgment operate as follows: the first rule expresses the fact that a normalized term of base type B does not contain subexpressions with nested collection type, therefore it can be shredded to itself, leaving the shredding environment Φ unchanged. This rule is also applied for plain aggregation, despite the fact that they have a collection subterm: from the grammar of flat relational normal forms, we know that such aggregatins must be in the form $\alpha(R^*)$, where R^* is a flat collection. Since R^* is flat, it does not need to be shredded recursively.

In the case of rows, we perform shredding pointwise on each field, connecting the input and output shredding environments in a pipeline, and finally combining together the shredded subterms in the obvious way.

The shredding of set and bag unions uses the query lifting technique and is performed by recursion on the subterms, using the same plumbing technique we employed for tuples; additionally, we optimize the output shredding environment by removing the graph queries $\overrightarrow{\psi}$ resulting from recursion, since they are absorbed into the new graph φ.

Finally, groupwise aggregation employs query lifting as well, since it returns a finite map. Since we know that the input is a flat relation, we do not need to shred the input query recursively.

$$
\begin{aligned}
([\,])^{\mathsf{sql}} &= \texttt{SELECT}\ 42\ \texttt{WHERE}\ 0 = 1 & (x.\ell)^{\mathsf{sql}} &= x.\ell \\
\left(c(\overrightarrow{X})\right)^{\mathsf{sql}} &= (c)^{\mathsf{sql}}\ (\overrightarrow{(X)^{\mathsf{sql}}}) & (\alpha(M))^{\mathsf{sql}} &= \texttt{SELECT}\ (\alpha)^{\mathsf{sql}}\ (x)\ \texttt{FROM}\ \left(\left[[\langle \bullet = z\rangle]^{\mathsf{bag}} \mid z \leftarrow M\right]^{\mathsf{bag}}\right)^{\mathsf{sql}}\ x \\
\left(\overrightarrow{\ell = X}\right)^{\mathsf{sql}} &= (X_1)^{\mathsf{sql}}\ \texttt{AS}\ \ell_1, \ldots, (X_n)^{\mathsf{sql}}\ \texttt{AS}\ \ell_n \\
\left(\overrightarrow{\ell = X}\right)^{\mathsf{sql}}_K &= (X_1)^{\mathsf{sql}}\ \texttt{AS}\ 1@\ell_1, \ldots, (X_n)^{\mathsf{sql}}\ \texttt{AS}\ 1@\ell_n & \left(\overrightarrow{\ell = X}\right)^{\mathsf{sql}}_V &= (X_1)^{\mathsf{sql}}\ \texttt{AS}\ 2@\ell_1, \ldots, (X_n)^{\mathsf{sql}}\ \texttt{AS}\ 2@\ell_n \\
\left(\bigcup \overrightarrow{C}\right)^{\mathsf{sql}} &= (C_1)^{\mathsf{sql}}\ \texttt{UNION}\ \ldots\ \texttt{UNION}\ (C_n)^{\mathsf{sql}} & \left(\biguplus \overrightarrow{D}\right)^{\mathsf{sql}} &= (D_1)^{\mathsf{sql}}\ \texttt{UNION ALL}\ \ldots\ \texttt{UNION ALL}\ (D_n)^{\mathsf{sql}} \\
(t)^{\mathsf{sql}} &= \texttt{SELECT}\ \left(\overrightarrow{\ell = x.\ell}\right)^{\mathsf{sql}}_V\ \texttt{FROM}\ t \quad (\text{if } t : \left[\langle \overrightarrow{\ell : T}\rangle\right]^{\mathsf{bag}}) \\
(\delta t)^{\mathsf{sql}} &= \texttt{SELECT DISTINCT}\ *\ \texttt{FROM}\ (t)^{\mathsf{sql}} & (\iota(Q))^{\mathsf{sql}} &= (Q)^{\mathsf{sql}} \\
(Q)^{\mathsf{sql}}_{GK} &= \texttt{SELECT DISTINCT}\ keys(Q, z)\ \texttt{FROM}\ (Q)^{\mathsf{sql}}\ z \\
\left(\gamma^{\mathsf{set}}_{x.\rho}(N)\right)^{\mathsf{sql}} &= \texttt{SELECT DISTINCT}\ (\rho)^{\mathsf{sql}}_K, vals(N, x)\ \texttt{FROM}\ (N)^{\mathsf{sql}}\ x \\
\left(\gamma^{\mathsf{set}}_{x.\rho}(N)\right)^{\mathsf{sql}}_{GK} &= \texttt{SELECT DISTINCT}\ (\rho)^{\mathsf{sql}}_V\ \texttt{FROM}\ (N)^{\mathsf{sql}}\ x \\
\left(\gamma^{\mathsf{bag}}_{x.\rho}(O)\right)^{\mathsf{sql}} &= \texttt{SELECT}\ (\rho)^{\mathsf{sql}}_K, vals(O, x)\ \texttt{FROM}\ (O)^{\mathsf{sql}}\ x \quad (\text{if } O : \left[\langle \overrightarrow{\ell : T}\rangle\right]^{\mathsf{bag}}) \\
\left(\gamma^{\mathsf{bag}}_{x.\rho}(O)\right)^{\mathsf{sql}}_{GK} &= \texttt{SELECT DISTINCT}\ (\rho)^{\mathsf{sql}}_V\ \texttt{FROM}\ (O)^{\mathsf{sql}}\ x \\
\left(N \circledast^{\mathsf{set}} \langle \overrightarrow{\Bbbk_i = M_i}\rangle\right)^{\mathsf{sql}} &= \texttt{SELECT DISTINCT}\ vals(N, x)\ \texttt{FROM}\ (N)^{\mathsf{sql}}\ x\ \texttt{WHERE}\ (\overrightarrow{x.1@\Bbbk_i = (M_i)^{\mathsf{sql}}}) \\
\left(O \circledast^{\mathsf{bag}} \langle \overrightarrow{\Bbbk_i = M_i}\rangle\right)^{\mathsf{sql}} &= \texttt{SELECT}\ vals(O, x)\ \texttt{FROM}\ (O)^{\mathsf{sql}}\ x\ \texttt{WHERE}\ (\overrightarrow{x.1@\Bbbk_i = (M_i)^{\mathsf{sql}}}) \\
\left(aggBy^{\mathsf{set}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}}(Q)\right)^{\mathsf{sql}} &= \texttt{SELECT}\ keys(Q, x), \overrightarrow{(\alpha)^{\mathsf{sql}}\ (x.2@\ell')\ \texttt{AS}\ 2@\ell}\ \texttt{FROM}\ (Q)^{\mathsf{sql}}\ x\ \texttt{GROUP BY}\ keys(Q, x) \\
\left(aggBy^{\mathsf{bag}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}}(R)\right)^{\mathsf{sql}} &= \texttt{SELECT}\ keys(R, x), \overrightarrow{(\alpha)^{\mathsf{sql}}\ (x.2@\ell')\ \texttt{AS}\ 2@\ell}\ \texttt{FROM}\ (R)^{\mathsf{sql}}\ x\ \texttt{GROUP BY}\ keys(R, x) \\
\left(\overrightarrow{F}, x \leftarrow N\right)^{\mathsf{sql}} &= \begin{cases} \left(\overrightarrow{F}\right)^{\mathsf{sql}}, ((N)^{\mathsf{sql}})\ x & (N \text{ closed wrt. } vars(F)) \\ \left(\overrightarrow{F}\right)^{\mathsf{sql}}, \texttt{LATERAL}\ ((N)^{\mathsf{sql}})\ x & (\text{otherwise}) \end{cases} \\
\left(\overrightarrow{F}, x \overset{\mathcal{K}}{\leftarrow} N\right)^{\mathsf{sql}} &= \begin{cases} (F)^{\mathsf{sql}}, ((N)^{\mathsf{sql}}_{GK})\ x & (N \text{ closed wrt. } vars(F)) \\ (F)^{\mathsf{sql}}, \texttt{LATERAL}\ ((F)^{\mathsf{sql}}_{GK})\ x & (\text{otherwise}) \end{cases} \\
\left(\overrightarrow{G}, x \leftarrow O\right)^{\mathsf{sql}} &= \begin{cases} \left(\overrightarrow{G}\right)^{\mathsf{sql}}, ((O)^{\mathsf{sql}})\ x & (O \text{ closed wrt. } vars(G)) \\ \left(\overrightarrow{G}\right)^{\mathsf{sql}}, \texttt{LATERAL}\ ((O)^{\mathsf{sql}})\ x & (\text{otherwise}) \end{cases} \\
\left(\overrightarrow{G}, x \overset{\mathcal{K}}{\leftarrow} Q^*\right)^{\mathsf{sql}} &= \begin{cases} \left(\overrightarrow{G}\right)^{\mathsf{sql}}, ((Q^*)^{\mathsf{sql}}_{GK})\ x & (Q^* \text{ closed wrt. } vars(G)) \\ \left(\overrightarrow{G}\right)^{\mathsf{sql}}, \texttt{LATERAL}\ ((G)^{\mathsf{sql}}_{GK})\ x & (\text{otherwise}) \end{cases}
\end{aligned}
$$

$$
\left(\left[[\rho \triangleright \langle \rho' \rangle]^{\mathsf{set}}\ \mathbf{where}^{\mathsf{set}}\ X \mid \overrightarrow{F}\right]^{\mathsf{set}}\right)^{\mathsf{sql}} = \texttt{SELECT DISTINCT}\ (\rho)^{\mathsf{sql}}_K,\ (\rho')^{\mathsf{sql}}_V\ \texttt{FROM}\ \left(\overrightarrow{F}\right)^{\mathsf{sql}}\ \texttt{WHERE}\ (X)^{\mathsf{sql}}
$$

$$
\left(\left[[\rho \triangleright \langle \rho' \rangle]^{\mathsf{bag}}\ \mathbf{where}^{\mathsf{bag}}\ X \mid \overrightarrow{G}\right]^{\mathsf{bag}}\right)^{\mathsf{sql}} = \texttt{SELECT}\ (\rho)^{\mathsf{sql}}_K, (\rho')^{\mathsf{sql}}_V\ \texttt{FROM}\ \overrightarrow{(G)^{\mathsf{sql}}}\ \texttt{WHERE}\ (X)^{\mathsf{sql}}
$$

$$
keys(Q, z) = z.1@\Bbbk_1\ \texttt{AS}\ 2@\Bbbk_1, \ldots, z.1@\Bbbk_n\ \texttt{AS}\ 2@\Bbbk_n \qquad (\text{if } Q : [T]^{\mathsf{set}}_{\overrightarrow{\Bbbk : S}} \text{ or } [T]^{\mathsf{bag}}_{\overrightarrow{\Bbbk : S}})
$$

$$
vals(Q, z) = z.2@\ell_1, \ldots, z.2@\ell_n \qquad (\text{if } Q : \left[\langle \overrightarrow{\ell : T}\rangle\right]^{\mathsf{set}}_P \text{ or } \left[\langle \overrightarrow{\ell : T}\rangle\right]^{\mathsf{bag}}_P)
$$

Figure 10. Translation to SQL

$$\frac{}{\Phi; \Theta \vdash B \Mapsto B \mid \Phi}$$

$$\frac{(\Phi_{i-1}; \Theta \vdash M_i \Mapsto \check{M}_i \mid \Phi_i)_{i=1,\ldots,n}}{\Phi_0; \Theta \vdash \overrightarrow{\ell = M} \Mapsto \overrightarrow{\ell = \check{M}} \mid \Phi_n}$$

$$\frac{\varphi \notin dom(\Phi_n) \qquad (\Phi_{i-1}; \Theta \vdash C_i \Mapsto \psi_i \circledast^{\mathrm{set}} \langle dom(\Theta) \rangle \mid \Phi_i)_{i=1,\ldots,n}}{\begin{array}{ll} \Phi_0; \Theta \vdash & \bigcup \overrightarrow{C} \Mapsto \varphi \circledast^{\mathrm{set}} \langle dom(\Theta) \rangle \\ \mid & (\Phi_n \setminus \overrightarrow{\psi})[\varphi \mapsto \bigcup \overrightarrow{\Phi_n(\psi)}] \end{array}}$$

$$\frac{\varphi \notin dom(\Phi_n) \qquad (\Phi_{i-1}; \Theta \vdash D_i \Mapsto \psi_i \circledast^{\mathrm{bag}} \langle dom(\Theta) \rangle \mid \Phi_i)_{i=1,\ldots,n}}{\begin{array}{ll} \Phi_0; \Theta \vdash & \biguplus \overrightarrow{D} \Mapsto \varphi \circledast^{\mathrm{bag}} \langle dom(\Theta) \rangle \\ \mid & (\Phi_n \setminus \overrightarrow{\psi})[\varphi \mapsto \biguplus \overrightarrow{\Phi_n(\psi)}] \end{array}}$$

$$\frac{\varphi \notin dom(\Psi) \qquad \Phi; \Theta, \overrightarrow{F} \vdash \rho \Mapsto \check{\rho} \mid \Psi}{\begin{array}{ll} \Phi; \Theta \vdash & \left[[\rho^* \triangleright \langle \rho \rangle]^{\mathrm{set}} \ \mathbf{where}\ B \mid \overrightarrow{F}\right]^{\mathrm{set}} \Mapsto \varphi \circledast^{\mathrm{set}} \langle dom(\Theta) \rangle \\ \mid & \Psi[\varphi \mapsto \mathcal{G}(\Theta; \left[[\rho^* \triangleright \check{\rho}]^{\mathrm{set}} \ \mathbf{where}\ B | \overrightarrow{F}\right]^{\mathrm{set}})] \end{array}}$$

$$\frac{\varphi \notin dom(\Psi) \qquad \Phi; \Theta, \overrightarrow{G^{\delta}} \vdash \rho \Mapsto \check{\rho} \mid \Psi}{\begin{array}{ll} \Phi_0; \Theta \vdash & \left[[\rho^* \triangleright \rho]^{\mathrm{bag}} \ \mathbf{where}\ B | \overrightarrow{G}\right]^{\mathrm{bag}} \Mapsto \varphi \circledast^{\mathrm{bag}} \langle dom(\Theta) \rangle \\ \mid & \Psi[\varphi \mapsto \mathcal{G}(\Theta; \left[[\rho^* \triangleright \check{\rho}]^{\mathrm{bag}} \ \mathbf{where}\ B | \overrightarrow{G}\right]^{\mathrm{bag}})] \end{array}}$$

$$\frac{\varphi \notin dom(\Phi)}{\begin{array}{ll} \Phi; \Theta \vdash & aggBy^{\mathrm{set}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}} (Q^*) \Mapsto \varphi \circledast^{\mathrm{set}} \langle dom(\Theta) \rangle \\ \mid & \Phi[\varphi \mapsto \mathcal{G}(\Theta; aggBy^{\mathrm{set}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}} (Q^*))] \end{array}}$$

$$\frac{\varphi \notin dom(\Phi)}{\begin{array}{ll} \Phi; \Theta \vdash & aggBy^{\mathrm{bag}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}} (R^*) \Mapsto \varphi \circledast^{\mathrm{bag}} \langle dom(\Theta) \rangle \\ \mid & \Phi[\varphi \mapsto \mathcal{G}(\Theta; aggBy^{\mathrm{bag}}_{\overrightarrow{z.\ell = \alpha(z.\ell')}} (R^*))] \end{array}}$$

$$(x \leftarrow O)^{\delta} \triangleq \begin{cases} x \leftarrow Q^* & \text{if } O = \iota Q^* \\ x \leftarrow \delta O & \text{else} \end{cases} \qquad \Phi \setminus \overrightarrow{\psi} \triangleq [(\varphi \mapsto N) \in \Phi \mid \varphi \notin \overrightarrow{\psi}]$$

$$(k \overset{\mathcal{K}}{\leftarrow} Q^*)^{\delta} \triangleq Q^*$$

Figure 11. Shredding rules.