Addressing an intensional expressiveness gap of comprehension syntax

LIMSOON WONG, National University of Singapore, Singapore

Comprehension syntax is widely adopted in modern programming languages as a means for manipulating collection types. This paper articulates and investigates an apparent gap in the intensional expressive power between comprehension syntax and relational database systems: (i) All subquadratic algorithms which are expressible in comprehension syntax, even after allowing some functions commonly available in the collection-type libraries of modern programming languages, do not compute low-selectivity joins. As database systems support these joins efficiently, this confirms the intensional expressiveness gap. (ii) A "Synchrony iterator" construct for synchronized iteration on multiple collections is introduced. This enables more algorithms, but not functions, to become definable using comprehension syntax. In particular, efficient algorithms for low-selectivity joins become expressible. So, the ability to iterate on multiple collections in synchrony constitutes an exact characterization of this intensional expressiveness gap. (iii) The proof of this intensional expressiveness gap relies on a "limited mixing" lemma which states that subquadratic algorithms expressible using comprehension syntax have limited ability for mixing atomic objects in their inputs. This limited-mixing lemma is non-query specific and is applicable even when ordered data types are present. It thus considerably enriches the available theoretical tools for studying intensional expressive power, as these tools are often query specific and are inapplicable in the presence of ordered data types. It is also a useful intensional counterpart to Gaifman's locality property. Gaifman's locality are very useful for analyzing extensional expressiveness of first-order query languages on unordered data types, but is not useful on ordered data types. (iv) Incidentally, efficient interval joins with overlap predicates are obtained as a free byproduct of Synchrony iterator. This kind of joins are often needed for practical applications such as temporal data and genomic data processing, but are not supported well in typical relational database systems.

CCS Concepts: • **Information systems → Relational database query languages**.

Additional Key Words and Phrases: Intensional expressive power, synchronized iteration, comprehension syntax

ACM Reference Format:
Limsoon Wong. 2022. Addressing an intensional expressiveness gap of comprehension syntax. *J. ACM* 1, 1 (January 2022), 35 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 OVERVIEW

Query languages based on comprehension syntax are able to express all relational queries supported by typical database systems [4, 23]. Moreover, queries written in comprehension syntax are appealingly simple [3]. So, comprehension syntax has become widely regarded as a means for embedding collection-type querying capabilities into programming languages. However, join queries expressed in comprehension syntax are generally compiled into nested loops; this implies

Author's address: Limsoon Wong, National University of Singapore, School of Computing, 13 Computing Drive, Singapore, 117417, Singapore, wongls@comp.nus.edu.sg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
0004-5411/2022/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

such queries typically have quadratic or even higher time complexity when they are expressed by means of comprehension syntax.

In contrast, in relational database systems, even in the absence of indices, when joins have low selectivity, these joins often have $O(n \log n)$ time complexity based on (sort-)merge-join algorithms [2]. And when the input relations are pre-sorted on their join attributes in a low-selectivity join, a merge-join can even be realised with linear time complexity by skipping the sorting steps.

Therefore, there is a potential intensional expressiveness gap between algorithms that can be realised by comprehension syntax and those used in database systems, viz. algorithms for low-selectivity joins. The first objective of this paper is to prove that this intensional expressiveness gap indeed exists. The proof goes via a "limited-mixing" lemma on $\mathcal{NRC}_1(\leq)$. On ordered data types, $\mathcal{NRC}_1(\leq)$ is equivalent to the flat relational algebra or first-order logic [23]. More pertinently, there is a simple translation between comprehension syntax and $\mathcal{NRC}_1(\leq)$, and this translation preserves time complexity. This makes $\mathcal{NRC}_1(\leq)$ a suitable ambient query language for investigating the potential intensional expressiveness gap between comprehension syntax and typical database systems. The limited-mixing lemma states that all $\mathcal{NRC}_1(\leq)$ queries of subquadratic time complexity are only able to mix atoms in their input relations in very limited ways. So, these subquadratic-complexity queries cannot be low-selectivity joins.

The second objective of this paper is to prove that this intensional expressiveness gap remains even when $\mathcal{NRC}_1(\leq)$ is further augmented by some common functions in typical programming language libraries. In particular, a limited-mixing lemma is proved when the functions *takewhile* and *dropwhile*, as well as a function for "instantaneous sorting" (a fictitious superfast constant-time sorting for arbitrary data), are added to $\mathcal{NRC}_1(\leq)$. The functions *takewhile* and *dropwhile* are considered because the iteration construct in $\mathcal{NRC}_1(\leq)$ must always access all elements in the relation being iterated on; whereas, *takewhile* and *dropwhile* are iteration functions which can stop their iterations mid-way according to some specified conditions. Due to the limited-mixing lemma on $\mathcal{NRC}_1(\leq, \textit{takewhile}, \textit{dropwhile}, \textit{sort})$, augmenting comprehension syntax with *takewhile* and *dropwhile* still does not permit low-selectivity joins at subquadratic time complexity to be realized, even in the presence of instantaneous sorting. So, the intensional expressiveness gap of comprehension syntax is not due solely to $\mathcal{NRC}_1(\leq)$'s lack of ability to stop an iteration mid way.

Another limited-mixing lemma is proved when the function *fold* is added to $\mathcal{NRC}_1(\leq, \textit{sort})$. *fold* is another function commonly provided in programming language libraries. *fold* is considered because it corresponds to structural recursion. This popular workhorse of functional programming languages tremendously expands the extensional expressive power of $\mathcal{NRC}_1(\leq)$. For example, transitive closure is expressible in $\mathcal{NRC}_1(\leq, \textit{fold})$ [4, 20], but not in $\mathcal{NRC}_1(\leq)$ [9]. The limited-mixing lemma on $\mathcal{NRC}_1(\leq, \textit{fold}, \textit{sort})$ means that augmenting comprehension syntax with *fold* still cannot realize low-selectivity joins at subquadratic time complexity, even in the presence of instantaneous sorting. So, the intensional expressiveness gap of comprehension syntax is not due solely to $\mathcal{NRC}_1(\leq)$'s lack of brute-force horsepower either.

One last limited-mixing lemma is proved when the function *zip* is added to $\mathcal{NRC}_1(\leq, \textit{sort})$. *zip* is arguably the most typical function in programming language libraries that simultaneously iterates on two collections. However, it is a rather limited form of synchronized iteration as it pairs objects in the two collections strictly by the physical positions, viz. first to first, second to second, and so on. As a result, it is hard to use *zip* to express joins. The limited-mixing lemma on $\mathcal{NRC}_1(\leq, \textit{zip}, \textit{sort})$ confirms the limitations of *zip* as a means for expressing efficient low-selectivity joins.

One might think that this intensional expressiveness gap is due to $\mathcal{NRC}_1(\leq)$'s simultaneous lack of an ability to stop an iteration mid-way and limitation in extensional expressive power. Indeed, low-selectivity joins can be expressed with subquadratic time complexity using *takewhile*, *dropwhile*, and *fold* simultaneously. However, while $\mathcal{NRC}_1(\leq, \textit{takewhile}, \textit{dropwhile}, \textit{fold}, \textit{sort})$ is able to express efficient algorithms for low-selectivity joins, it can also express many other functions (e.g., transitive closure, and even exponential-time queries when arithmetics operators are available), which are inexpressible in $\mathcal{NRC}(\leq)$. In other words, the ability to stop an iteration mid-way and to perform structural recursion does not constitute a precise characterization of the intensional expressiveness gap of comprehension syntax.

The third and final objective of this paper is to more precisely characterize this gap. The constructs *eiterator* and *syncedwith* for synchronized iteration on two or more relations are introduced. Efficient merge-joins are expressible in $\mathcal{NRC}_1(\leq,$ *eiterator*, *syncedwith*), despite $\mathcal{NRC}_1(\leq)$ and $\mathcal{NRC}_1(\leq,$ *eiterator*, *syncedwith*) having the same extensional expressive power. Therefore, the contribution of *eiterator* and *syncedwith* is solely to intensional expressive power, permitting the expression of a richer repertoire of algorithms but not functions. In other words, synchronized iteration more precisely characterizes the intensional expressiveness gap between comprehension syntax and typical database systems. Moreover, as it will be appreciated later, *eiterator* and *syncedwith* syntactically dovetail into comprehension syntax. So, efficient joins expressed in $\mathcal{NRC}_1(\leq,$ *eiterator*, *syncedwith*) often do not look syntactically too different from the inefficient versions in $\mathcal{NRC}_1(\leq)$. Hence, comprehension syntax with *eiterator* and *syncedwith* can be claimed as a more genuinely natural embedding of collection-type querying capabilities into programming languages in terms of both extensional and intensional expressive power. Also, remarkably, efficient interval joins with overlap predicates which are not well supported by typical database systems come for free in $\mathcal{NRC}_1(\leq,$ *eiterator*, *syncedwith*).

2 NESTED RELATIONAL CALCULUS

The restriction of the nested relational calculus $\mathcal{NRC}$ from Buneman et al. [4] and Wong [23] to flat relations is used as the ambient language here. $\mathcal{NRC}$ is equivalent to the usual nested relational algebra [4, 23]. Its restriction to flat relations, denoted here as $\mathcal{NRC}_1$, is equivalent to flat relational algebra and first-order logic [23]. This ambient language, and its operational semantics and rewrite rules, are described below.

2.1 Types and expressions

The types and expressions of $\mathcal{NRC}$ are given in Figure 1. The type superscripts in the figure are omitted when there is no confusion. For simplicity, all variable names are assumed to be distinct. For convenience, all data types are endowed with an order; this query language is denoted as $\mathcal{NRC}(\leq)$.

The semantics of a type is just a set of objects built up by nesting sets and records of base-type objects. Base types are denoted by b (representing atomic values in a database). An object of type $s_1 \times \cdots \times s_n$ is a tuple (i.e., a record) whose ith component is an object of type s_i, for $1 \leq i \leq n$. An object of type $\{s\}$ is a finite set whose elements are objects of type s; an object of type $\{s\}$ is called a set or a "relation." Moreover, if $s = b \times \cdots \times b$, then an object of type $\{s\}$ (or s) is called a "flat relation." However, if s contains some set brackets, then an object of type $\{s\}$ is called a "nested relation."

The expression constructs are defined as follows. The expression C denotes objects, including constants of base types b; the syntax for C will be given in the next subsection. The expression $(e_1, \ldots, e_n)$ forms a tuple whose ith component is the object denoted by e_i, for $1 \leq i \leq n$. The expression $e.\pi_i$ extracts the ith component of the tuple e. The expressions $\{\}$, $\{e\}$, and $e_1 \cup e_2$ have their conventional meaning as set operations. The expression $\bigcup\{e_1 \mid x \in e_2\}$ forms the set obtained by first applying the function $f(x) = e_1$ to each object in the set e_2 and then taking their union; that is, $\bigcup\{e_1 \mid x \in e_2\} = f(C_1) \cup \ldots \cup f(C_n)$, where $f(x) = e_1$ and $\{C_1, \ldots, C_n\}$ is the set denoted by e_2.

Types in $\mathcal{NRC}$

$$s ::= b \mid s_1 \times \cdots \times s_n \mid \{s\}$$
where b is a base type.

Expressions in $\mathcal{NRC}$

$$\frac{}{C^s : s} \qquad \frac{}{x^s : s} \qquad \frac{e_1 : s_1 \quad \ldots \quad e_n : s_n}{(e_1, \ldots, e_n) : s_1 \times \cdots \times s_n} \qquad \frac{e : s_1 \times \cdots \times s_n}{e.\pi_i : s_i} 1 \leq i \leq n$$

$$\frac{}{\{\}^s : \{s\}} \qquad \frac{e : s}{\{e\} : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \{s\}}{e_1 \cup e_2 : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \{t\}}{\bigcup\{e_1 \mid x^t \in e_2\} : \{s\}}$$

$$\frac{}{\mathit{true} : \mathbb{B}} \qquad \frac{}{\mathit{false} : \mathbb{B}} \qquad \frac{e_1 : \mathbb{B} \quad e_2 : s \quad e_3 : s}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : s}$$

$$\frac{e_1 : s \quad e_2 : s}{e_1 < e_2 : \mathbb{B}} \qquad \frac{e_1 : s \quad e_2 : s}{e_1 = e_2 : \mathbb{B}} \qquad \frac{e : \{s\}}{e\ \mathit{isempty} : \mathbb{B}}$$

Fig. 1. $\mathcal{NRC}$.

Besides the object types and their expression constructs above, $\mathcal{NRC}$ also has the Boolean type $\mathbb{B}$ as a base type, and the expression constructs *true*, *false*, and *if* e_1 *then* e_2 *else* e_3, which have their conventional meaning as Boolean values and conditional expression. Lastly, the expression $e_1 < e_2$ provides a linear ordering on objects of the same type; the expression $e_1 = e_2$ checks whether the objects denoted by e_1 and e_2 are the same; and the expression e *isempty* checks whether the set denoted by e is empty.

The emptiness test e *isempty*, the equality test $e_1 = e_2$, and the ordering test $e_1 < e_2$ are provided for every type s solely for convenience. Actually, they are definable in terms of the tests on base types b. In particular, the linear ordering on any arbitrarily deeply nested combinations of record and set types can be lifted—in a manner definable by $\mathcal{NRC}$—from the linear ordering on each base type b as follows [14]: for tuple types $s_1 \times \cdots \times s_n$, it is defined pointwise lexicographically; and for set types $\{s\}$, it is defined a la Wechler [22] based on the Hoare ordering (viz. $X \leq Y$ iff for all $x \in X - Y$, there is $y \in Y - X$, such that $x \leq y$.)

The notation $x \in e_2$ in the $\bigcup\{e_1 \mid x \in e_2\}$ construct is an abstraction that introduces the variable x whose scope is the expression e_1. That is, it is part of the syntax and is not a membership test. This construct is the sole means in $\mathcal{NRC}$ for iterating over a set.

If a variable appearing in an expression e is not introduced by a subexpression of the form $\bigcup\{e_1 \mid x \in e_2\}$ in e, it is called a free variable of e. When it is necessary to explicitly indicate the free variables of an expression, we write $e(x_1, ..., x_2)$ or $e(\vec{x})$. An expression $e(\vec{x})$, with free variables $\vec{x}$ can be regarded as a function $f(\vec{x}) = e(\vec{x})$. When it is desirable to distinguish the free variables local to a subexpression $e(\vec{x}, \vec{X})$ of an expression $e'(\vec{X})$, uppercase is used for the free variables of the entire expression while lowercase is used for other free variables of the subexpression. Also, an expression e having no free variable is called a closed expression.

When objects $\vec{C}$ have the same types as the free variables $\vec{x}$ of an expression $e(\vec{x})$, the expression obtained by replacing each variable x_i in $\vec{x}$ in $e(\vec{x})$ by the corresponding C_i in $\vec{C}$ is denoted as $e[\vec{C}/\vec{x}]$. The result of applying $e(\vec{x})$ as a function to $\vec{C}$ is denoted by $e(\vec{C})$. To make notations lighter, $e(\vec{C})$ is sometimes also used to denote the expression $e[\vec{C}/\vec{x}]$; however, this usage is generally eschewed in proofs. Sometimes, when it is unimportant to know what the objects $\vec{C}$ are, the notation $e(\vec{x})$ is used instead of writing $e(\vec{C})$ explicitly.

A "pattern-matching" construct $\bigcup\{e_1 \mid (x_1, \ldots, x_n) \in e_2\}$ is used for convenience. It is a syntactic sugar for $\bigcup\{e_1[x.\pi_1/x_1, \ldots, x.\pi_n/x_n] \mid x \in e_2\}$. There is also an easy mechanical translation [3, 23] between the syntax of $\mathcal{NRC}$ and comprehension syntax of the form $\{e \mid \delta_1, \ldots, \delta_n\}$ where each δ_i either has the form $\vec{x}_i \in e_i$ or the form e_i. The translation is as follows:

- $\{e \mid \vec{x}_1 \in e_1, \Delta\} =_{df} \bigcup\{\{e \mid \Delta\} \mid \vec{x}_1 \in e_1\}$;
- $\{e \mid e_1, \Delta\} =_{df}$ *if* e_1 *then* $\{e \mid \Delta\}$ *else* $\{\}$; and
- $\{e \mid \} =_{df} \{e\}$.

Comprehension syntax is used here to write examples, but the reader should understand these examples as syntactic sugars of the actual $\mathcal{NRC}$ expressions. Below are some examples.

Example 2.1. All relational queries [5] are expressible in $\mathcal{NRC}$.

- $\Pi_i\, X =_{df} \{x.\pi_i \mid x \in X\}$ is the relational projection;
- $\sigma_d\, X =_{df} \{x \mid x \in X, d(x)\}$ is the relational selection;
- $X \bowtie Y =_{df} \{(x, y) \mid (u, x) \in X, (v, y) \in Y, u = v\}$ is the relational join;
- $X \cap Y =_{df} \{x \mid x \in X,$ *not* $\{y \mid y \in Y, y = x\}$ *isempty*$\}$ is the relational intersection.
- $X - Y =_{df} \{x \mid x \in X, \{y \mid y \in Y, y = x\}$ *isempty*$\}$ is the relational difference; and
- $X \div Y =_{df} \{x \mid (x, y) \in X, Y \subseteq \{y' | (x', y') \in X, x' = x\}\}$, where $Y \subseteq Y' =_{df} Y - Y'$ *isempty*, is the relational division.

Example 2.2. $\mathcal{NRC}$ can also express nested relational operations [21].

- *unnest* $R =_{df} \{(x, y) | (X, y) \in R, x \in X\}$ unnests the nested relation R; and
- *nest* $R =_{df} \{(\{x \mid (x, y) \in R, y = v\}, v) \mid (u, v) \in R\}$ creates a nested version of a relation R, which groups values in the first column of R by values in the second column of R.

Let $\mathcal{NRC}_1$ denote the fragment of $\mathcal{NRC}$ where expressions are restricted to flat relation types. That is, in $\mathcal{NRC}_1$, every (sub)expression $e(x_1, ..., x_n) : s$ where $x_i : s_i$ for $1 \leq i \leq n$, the types s, s_1, ..., s_n are all flat relations. It is known that $\mathcal{NRC}$ enjoys the conservative extension property [23]; thus, $\mathcal{NRC}(\leq)$ and $\mathcal{NRC}_1(\leq)$ express the same functions on flat relations, and are equivalent to flat relational algebra or first-order logic with ordering $FO(\leq)$.

Proposition 2.3. *$\mathcal{NRC}(\leq)$, $\mathcal{NRC}_1(\leq)$, and $FO(\leq)$ have the same extensional expressive power on flat relations.*

An expression $e(\vec{x})$ in $\mathcal{NRC}$ can always be turned into an expression $e'(\vec{y}, \vec{x})$ such that no constants or objects appear in it. This can be obtained by introducing fresh free variables $\vec{y}$ and replacing each object C_i in $e(\vec{x})$ by the variable y_i; then $e'[\vec{C}/\vec{y}](\vec{x}) = e(\vec{x})$. So, for simplicity, and without loss of generality, only constant-free expressions are considered when results are stated and proved in this paper.

2.2 Operational semantics

In order to discuss intensional expressive power, i.e. what algorithms are expressible, it is necessary to know how an expression of $\mathcal{NRC}$ is executed. This is specified in Figure 2 as a call-by-value operational semantics. A call-by-value operational semantics is widely adopted in programming

$$\frac{}{\Gamma, C \Downarrow \Gamma, C}$$

$$\frac{\Gamma_0, e_1 \Downarrow \Gamma_1, C_1 \quad \ldots \quad \Gamma_{n-1}, e_n \Downarrow \Gamma_n, C_n}{\Gamma_0, (e_1, \ldots, e_n) \Downarrow \Gamma_n, (C_1, \ldots, C_n)} \qquad \frac{\Gamma, e \Downarrow \Gamma', (C_1, \ldots, C_n)}{\Gamma, e.\pi_i \Downarrow \Gamma', C_i} 1 \leq i \leq n$$

$$\frac{}{\Gamma, \{\} \Downarrow \Gamma, \{\}} \qquad \frac{\Gamma, e \Downarrow \Gamma', C}{\Gamma, \{e\} \Downarrow \Gamma', \{C\}} \qquad \frac{\Gamma, e_1 \Downarrow \Gamma_1, C_1 \quad \Gamma_1, e_2 \Downarrow \Gamma_2, C_2}{\Gamma, e_1 \cup e_2 \Downarrow \Gamma_2, C_1 \oplus C_2}$$

$$\frac{\begin{array}{c}\Gamma, e_2 \Downarrow \Gamma_0, \{C_1, \ldots, C_n\} \\ \Gamma_0, e_1[C_1/x] \Downarrow \Gamma_1, C_1' \quad \cdots \quad \Gamma_{n-1}, e_1[C_n/x] \Downarrow \Gamma_n, C_n'\end{array}}{\Gamma, \bigcup\{e_1 \mid x \in e_2\} \Downarrow \Gamma_n, C_1' \oplus \cdots \oplus C_n'}$$

$$\frac{}{\Gamma, \mathit{true} \Downarrow \Gamma, \mathit{true}} \qquad \frac{}{\Gamma, \mathit{false} \Downarrow \Gamma, \mathit{false}}$$

$$\frac{\Gamma, e_1 \Downarrow \Gamma_1, \mathit{true} \quad \Gamma_1, e_2 \Downarrow \Gamma_2, C}{\Gamma, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow \Gamma_2, C} \qquad \frac{\Gamma, e_1 \Downarrow \Gamma_1, \mathit{false} \quad \Gamma_1, e_3 \Downarrow \Gamma_3, C}{\Gamma, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow \Gamma_3, C}$$

$$\frac{\Gamma, e_1 \Downarrow \Gamma_1, C_1 \quad \Gamma_1, e_2 \Downarrow \Gamma_2, C_2}{\Gamma, e_1 < e_2 \Downarrow \Gamma_2, \mathit{true}} C_1 < C_2 \qquad \frac{\Gamma, e_1 \Downarrow \Gamma_1, C_1 \quad \Gamma_1, e_2 \Downarrow \Gamma_2, C_2}{\Gamma, e_1 < e_2 \Downarrow \Gamma_2, \mathit{false}} C_1 \not< C_2$$

$$\frac{\Gamma, e_1 \Downarrow \Gamma_1, C_1 \quad \Gamma_1, e_2 \Downarrow \Gamma_2, C_2}{\Gamma, e_1 = e_2 \Downarrow \Gamma_2, \mathit{true}} C_1 = C_2 \qquad \frac{\Gamma, e_1 \Downarrow \Gamma_1, C_1 \quad \Gamma_1, e_2 \Downarrow \Gamma_2, C_2}{\Gamma, e_1 = e_2 \Downarrow \Gamma_2, \mathit{false}} C_1 \neq C_2$$

$$\frac{\Gamma, e \Downarrow \Gamma', C}{\Gamma, e\ \mathit{isempty} \Downarrow \Gamma', \mathit{true}} C = \{\} \qquad \frac{\Gamma, e \Downarrow \Gamma', C}{\Gamma, e\ \mathit{isempty} \Downarrow \Gamma', \mathit{false}} C \neq \{\}$$

Fig. 2. A call-by-value operational semantics of $\mathcal{NRC}$.

languages and has also been used for several variations of $\mathcal{NRC}$ in earlier works [19, 20, 25] on intensional expressive power.

In Figure 2, the notation $\Gamma, e \Downarrow \Gamma', C$ means the closed expression e is evaluated in the environment Γ to produce the object C and the updated environment Γ'. The operational semantics specified in the figure is free from side effects and thus does not update the environment; i.e. $\Gamma, e \Downarrow \Gamma', C$ implies $\Gamma = \Gamma'$. Hence, the definition of an environment is postponed to a later section, when expression constructs which update the environment are introduced.

The unique evaluation tree of e in the environment Γ is denoted using the notation $\Gamma, e \Downarrow$. The "step" complexity $\mathit{step}(\Gamma, e \Downarrow)$ of an evaluation is defined as the time complexity of the largest node in the evaluation tree—viz., $\mathit{step}(\Gamma, e \Downarrow) = \max\{\mathit{time}(\Gamma', e' \Downarrow \Gamma'', C') \mid$ the node $\Gamma', e' \Downarrow \Gamma'', C'$ occurs in the evaluation tree of $\Gamma, e \Downarrow\}$. The time complexity $\mathit{time}(\Gamma', e' \Downarrow \Gamma'', C')$ of a node is the number of branches that the node has. E.g., in Figure 2, $\mathit{time}(\Gamma, \bigcup\{e_1 \mid x \in e_2\} \Downarrow \Gamma_n, C_1' \oplus \cdots \oplus C_n') = n + 1$. On the other hand, the time complexity $\mathit{time}(\Gamma, e \Downarrow)$ of an evaluation is the sum of the time complexity of all the nodes in the tree. When Γ is obvious from the context, it is omitted from the notations above, viz. $e \Downarrow C$, $\mathit{step}(e \Downarrow)$, $\mathit{time}(e \Downarrow C)$, and $\mathit{time}(e \Downarrow)$.

The syntax for objects C is as follows. A constant c of a base type b is an object of type b. A tuple $(C_1, ..., C_n)$ is an object of type $s_1 \times \cdots \times s_n$ if each C_i is an object of type s_i. An "enumeration list", elist for short, $\{C_1, ..., C_n\}$ is an object of type $\{s\}$ if each C_i is an object of type s.

An elist $\{C_1, .., C_n\}$ can be thought of as a particular way of enumerating the elements of the set that it represents, viz. C_1 followed by C_2, followed by C_3, and so on. There are as many distinct elists that represent the same set as there are distinct ways to enumerate elements of that set, corresponding to different ordering and multiplicity of appearances of its elements in the enumeration.

The notations $C = C'$ and $C == C'$ are used to refer to two notions of equality involving elists. The notation $C = C'$ means C are C' are the same objects when all the elists contained in them (and objects therein) are interpreted as sets: thus, $c = c'$ iff c and c' are the same constant of a base type; $(C_1, ..., C_n) = (C'_1, ..., C'_n)$ iff $C_i = C'_i$ for $1 \leq i \leq n$; and $\{C_1, ..., C_n\} = \{C'_1, ..., C'_m\}$ iff for each $1 \leq i \leq n$, there is $1 \leq j \leq m$ such that $C_i = C'_j$, and for each $1 \leq j \leq m$, there is $1 \leq i \leq n$ such that $C_i = C'_j$. The notation $C == C'$ means C and C' are the same objects when all the elists contained in them (and objects therein) are interpreted as lists: thus, $c == c'$ iff c and c' are the same constant of a base type; $(C_1, ..., C_n) == (C'_1, ..., C'_n)$ iff $C_i == Ci'$ for $1 \leq i \leq n$; and $\{C_1, ..., C_n\} == \{C'_1, ..., C'_m\}$ iff $n = m$, and $C_i == C'_i$ for $1 \leq i \leq n$.

In Figure 2, a constructor $C \oplus C'$ is used to produce the concatenation of two elists in constant time; i.e. given $C == \{C_1, ..., C_n\}$ and $C' == \{C'_1, ..., C'_m\}$, $C \oplus C' == \{C_1, ..., C_n, C'_1, ..., C'_m\}$. Also, $\oplus$ is always used in a right-associative manner; e.g., $C \oplus C' \oplus C''$ means $C \oplus (C' \oplus C'')$. Note that while it is not a common practice to use a constant-time concatenation constructor to represent lists, it has been used in e.g. the influential Kleisli Query System [24] which is based on $\mathcal{NRC}$.

Linear orderings $<$ are available on all base types and are lifted to all types, as defined earlier. With this, the subset of objects in "canonical form" can be defined as follows. A constant c of any base type b is canonical. A tuple $(C_1, ..., C_n)$ is canonical if each C_i is canonical. An elist $\{C_1, ..., C_n\}$ is canonical if for every $1 \leq i, j \leq n$, it is the case that C_i is canonical, C_j is canonical, and $C_i < C_j$ iff $i < j$; a canonical elist is thus duplicate-free and is sorted according to $<$. The notation $canonize(C)$ denotes the unique canonical form of the object C. Clearly, for $C == \{C_1, ..., C_n\}$ representing a flat relation, $canonize(C)$ can be produced in $O(n \log(n))$ time.

The call-by-value operational semantics in Figure 2 does not perform canonization. This is because canonization is not needed to guarantee the soundness of an evaluation in $\mathcal{NRC}(\leq)$.

Proposition 2.4 (Soundness). *Suppose $e(\vec{x}) : s$ is an expression in $\mathcal{NRC}$, $\vec{C}$ are objects having the same types as $\vec{x}$, and $e[\vec{C}/\vec{x}] \Downarrow C'$. Then $e[\vec{C}/\vec{x}] = C'$.*

The size of an object C can be defined in any reasonable way. One way is to define $size(C)$ as the number of symbols used to write C out. Another way, when C is an elist, is to defined $size(C)$ as $|C|$, the length of the elist. Both notions of size can be generalized to $size(\vec{C}) = \sum_i size(C_i)$. The latter notion of input size is used by default. Then the time complexity of an expression $e(\vec{x})$ can be defined in the usual way based on input size; i.e. $time(e(\vec{x})) = \Theta(g(n))$ where n denotes input size and g is a function of n. It is easily shown that all queries in $\mathcal{NRC}$ have polynomial time complexity [4].

Proposition 2.5 (Polynomiality). *Let $e(\vec{x}) : s$ be an expression in $\mathcal{NRC}(\leq)$. Then there is a number k such that $time(e(\vec{x})) = \Theta(n^k)$ where n denotes input size. In particular, if $time(e(\vec{x}))$ is sub-quadratic, then it must be either linear or constant time; and if it is sub-linear, then it must be constant time. Furthermore, this polynomiality property is retained when $\mathcal{NRC}$ is augmented by any additional constructs that have polynomial time complexity.*

$$
\begin{array}{rcl}
\bigcup\{e \mid x \in \{\}\} & \mapsto & \{\} \\
\bigcup\{e_1 \mid x \in \{e_2\}\} & \mapsto & e_1[e_2/x] \\
\bigcup\{e \mid x \in (e_1 \cup e_2)\} & \mapsto & \bigcup\{e \mid x \in e_1\} \cup \bigcup\{e \mid x \in e_2\} \\
\bigcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\} & \mapsto & \bigcup\{\bigcup\{e_1 \mid x \in e_2\} \mid y \in e_3\} \\
\bigcup\{e \mid x \in (\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3)\} & \mapsto & \mathit{if}\ e_1\ \mathit{then}\ \bigcup\{e \mid x \in e_2\}\ \mathit{else}\ \bigcup\{e \mid x \in e_3\} \\
(e_1, \ldots, e_2).\pi_i & \mapsto & e_i \\
(\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3).\pi_i & \mapsto & \mathit{if}\ e_1\ \mathit{then}\ e_2.\pi_i\ \mathit{else}\ e_3.\pi_i \\
\mathit{if}\ \mathit{true}\ \mathit{then}\ e_2\ \mathit{else}\ e_3 & \mapsto & e_2 \\
\mathit{if}\ \mathit{false}\ \mathit{then}\ e_2\ \mathit{else}\ e_3 & \mapsto & e_3
\end{array}
$$

Fig. 3. A system of rewrite rules for $\mathcal{NRC}$.

2.3 Rewrite rules

Figure 3 shows a system of rewrite rules for simplifying $\mathcal{NRC}$ expressions. These rules have been used in many previous works on $\mathcal{NRC}$ [15, 16, 23, 25]. These rules are easily shown to be sound, and do not increase step complexity, and are strongly normalizing [23].

Although this system of rewrite rules does not increase step complexity, it can increase time complexity. E.g., rewriting $\bigcup\{\bigcup\{\{(x,z)\} \mid z \in Z\} \mid x \in \{\bigcup\{\{y.\pi_1\} \mid y \in Y\}\}\}$ to $\bigcup\{\{(\bigcup\{\{y.\pi_1\} \mid y \in Y\}, z)\} \mid z \in Z\}$ by the second rule in Figure 3, changes the time complexity from $O(|Y| + |Z|)$ to $O(|Z| * |Y|)$.

Fortunately, the second rule in Figure 3 is the only rule that misbehaves this way. For convenience of reference, the system of rewrite rules in Figure 3 is called the unrestricted system. And when the second rule is excluded, it is called the restricted system.

Proposition 2.6 (Normal form). *Let $e(\vec{X}) : s$ be an expression in $\mathcal{NRC}(\leq)$, and $\vec{C}$ be objects having the same types as $\vec{X}$.*

(1) $e[\vec{C}/\vec{X}] == e'[\vec{C}/\vec{X}]$ *if* $e \mapsto e'$.
(2) $\mathit{step}(e[\vec{C}/\vec{X}] \Downarrow) \geq \mathit{step}(e'[\vec{C}/\vec{X}] \Downarrow)$ *if* $e \mapsto e'$.
(3) $\mathit{time}(e[\vec{C}/\vec{X}] \Downarrow) \geq \mathit{time}(e'[\vec{C}/\vec{X}] \Downarrow)$ *if* $e \mapsto e'$ *under the restricted system of rewrite rules.*
(4) *The (un)restricted system of rewrite rules is strongly normalizing.*
(5) *The unrestricted system of rewrite rules induces a normal form, wherein every subexpression of the form* $\bigcup\{e_1(y, \vec{x}, \vec{X}) \mid y \in e_2(\vec{x}, \vec{X})\}$, $e_2(\vec{x}, \vec{X})$ *must be one of the variables in* $\vec{X}$.
(6) *The restricted system of rewrite rules induces a normal form, wherein every subexpression of the form* $\bigcup\{e_1(y, \vec{x}, \vec{X}) \mid y \in e_2(\vec{x}, \vec{X})\}$, $e_2(\vec{x}, \vec{X})$ *must be one of the variables in* $\vec{X}$ *or* $e_2(\vec{x}, \vec{X})$ *has the form* $\{e_3(\vec{x}, \vec{X})\}$.

3 A LIMITED-MIXING LEMMA

An analysis of the normal form induced by the restricted system of rewrite rules yields a useful limited-mixing lemma on $\mathcal{NRC}_1(\leq)$. The lemma is proved below, after some relevant definitions are given.

A level-0 atom of an object C is a constant c which has at least one occurrence in C that is not inside any elist in C. A level-1 atom of an object C is a constant c which has at least one occurrence in C that is inside an elist which is not nested inside another elist in C. All other constants appearing in an object C are higher level atoms. The notations $\mathit{atom}^0(C)$, $\mathit{atom}^1(C)$, and $\mathit{atom}^{\leq 1}(C)$ respectively denote the set of level-0 atoms of C, the set of level-1 atoms of C, and their union. The level-0 molecules of an object C are the elists in C that are not nested inside other elists. The notation $\mathit{molecule}^0(C)$ denotes the set of level-0 molecules of C. E.g., suppose $C = (c_1, c_2, \{(c_3, c_4, \{(c_5, c_6)\})\})$; then $\mathit{atom}^0(C) = \{c_1, c_2\}$, $\mathit{atom}^1(C) = \{c_3, c_4\}$, $\mathit{atom}^{\leq 1}(C) = \{c_1, c_2, c_3, c_4\}$, $\{c_5, c_6\}$ are higher-level atoms, and $\mathit{molecule}^0(C) = \{\{(c_3, c_4, \{(c_5, c_6)\})\}\}$.

The level-0 Gaifman graph of an object C is defined as an undirected graph $\mathit{gaifman}^0(C)$ whose nodes are the level-0 atoms of C, and edges are all the pairs of level-0 atoms of C. The level-1 Gaifman graph of an object C is defined as an undirected graph $\mathit{gaifman}^1(C)$ whose nodes are the level-1 atoms of C, and the edges are defined as follow: If $C == \{C_1, ..., C_n\}$, the edges are pairs (x, y) such that x and y are in the same $\mathit{atom}^0(C_i)$ for some $1 \leq i \leq n$; if $C == (C_1, ..., C_n)$, the edges are pairs $(x, y) \in \mathit{gaifman}^1(C_i)$ for some $1 \leq i \leq n$; and there are no other edges. The Gaifman graph [8] of an object C is defined as $\mathit{gaifman}(C) = \mathit{gaifman}^0(C) \cup \mathit{gaifman}^1(C)$.

It is shown below, by induction on the structure of $\mathcal{NRC}_1(\leq)$ expressions, that they manipulate their inputs in highly restricted local manners. In particular, expressions which have contant time complexity are unable to mix level-0 and level-1 atoms. And expressions which have linear time complexity are able to mix level-0 atoms with level-0 and level-1 atoms, but are unable to mix level-1 atoms with themselves or with higher-level atoms.

Lemma 3.1 (Limited mixing). *Let $e(\vec{X}) : s$ be an expression in $\mathcal{NRC}_1(\leq)$. Suppose objects $\vec{C}$ have the same types as $\vec{X}$, and $e[\vec{C}/\vec{X}] \Downarrow C'$.*

(1) *If $e(\vec{X})$ has constant time complexity, then*
(i) $\mathit{atom}^0(C') \subseteq \mathit{atom}^0(\vec{C})$,
(ii) $\mathit{atom}^1(C') \subseteq \mathit{atom}^{\leq 1}(\vec{C})$,
(iii) $\mathit{gaifman}(C') \subseteq \mathit{gaifman}(\vec{C})$,
(iv) for each $U \in \mathit{molecule}^0(C')$, there are V_0, V_1, ..., V_m such that $\mathit{atom}^1(V_0) \subseteq \mathit{atom}^0(\vec{C})$, $V_j \in \mathit{molecule}^0(\vec{C})$ for each $1 \leq j \leq m$, and $U = V_0 \cup V_1 \cup \cdots \cup V_m$.

(2) *If $e(\vec{X})$ has linear time complexity, then*
(i) $\mathit{atom}^0(C') \subseteq \mathit{atom}^0(\vec{C})$,
(ii) $\mathit{atom}^1(C') \subseteq \mathit{atom}^{\leq 1}(\vec{C})$, and
(iii) for each $(u, v) \in \mathit{gaifman}(C')$, either $(u, v) \in \mathit{gaifman}(\vec{C})$, or $u \in \mathit{atom}^0(\vec{C})$ and $v \in \mathit{atom}^1(\vec{C})$, or $u \in \mathit{atom}^1(\vec{C})$ and $v \in \mathit{atom}^0(\vec{C})$.

Proof. Part 1 of the lemma is straightforward; its proof is omitted. For Part 2, by Proposition 2.6, $e(\vec{X})$ is be assumed to be in the normal form induced by the restricted system of rewrite rules. The proof proceeds by structural induction on $e(\vec{X})$.

This first interesting case is when $e(\vec{X})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in X_0\}$, where X_0 is one of the free variables in $\vec{X}$, and has linear time complexity. Then $\mathit{time}(e_1(x, \vec{X}))$ must have constant time complexity; otherwise, the whole expression has quadratic time complexity. Let $\vec{C}$ have the types of $\vec{X}$ and let C_0 in $\vec{C}$ correspond to X_0. Suppose $C_x \in C_0$ and $e_1[C_x/x, \vec{C}/\vec{X}] \Downarrow C'_x$. As C'_x has set type, $\mathit{atom}^0(C'_x) = \{\} \subseteq \mathit{atom}^0(\vec{C})$. This proves Part 2(i). Since $C_x \in C_0$ and C_0 is in $\vec{C}$, $\mathit{atom}^0(C_x) \in \mathit{atom}^1(\vec{C})$. Also, as this lemma concerns $\mathcal{NRC}_1$, C_x must have type $b \times \cdots \times b$; thus, $\mathit{atom}^1(C_x) = \{\}$. So, by the induction hypothesis of Part 2, $\mathit{atom}^1(C'_x) \subseteq \mathit{atom}^{\leq 1}(C_x, \vec{C}) = \mathit{atom}^{\leq 1}(\vec{C})$. This proves Part 2(ii). As $e_1(x, \vec{X})$ has constant time complexity, and C_x has type $b \times \cdots \times b$, by the induction hypothesis of Part 1, we get $\mathit{gaifman}(C'_x) \subseteq \mathit{gaifman}(C_x, \vec{C}) = \mathit{gaifman}^0(C_x, \vec{C}) \cup \mathit{gaifman}^1(\vec{C})$. Suppose $(u, v) \in \mathit{gaifman}(C'_x)$. If $u \in \mathit{atom}^0(C_x)$ and $v \in \mathit{atom}^0(C_x)$, $(u, v) \in \mathit{gaifman}^1(\vec{C}) \subseteq \mathit{gaifman}(\vec{C})$. If $u \in \mathit{atom}^0(C_x)$ and $v \notin \mathit{atom}^0(C_x)$, $u \in \mathit{atom}^1(C_0) \subseteq \mathit{atom}^1(\vec{C})$ and $v \in \mathit{atom}^0(\vec{C})$. If $u \notin \mathit{atom}^0(C_x)$ and $v \in \mathit{atom}^0(C_x)$, $v \in \mathit{atom}^1(C_0) \subseteq \mathit{atom}^1(\vec{C})$ and $u \in \mathit{atom}^0(\vec{C})$. If $u \notin \mathit{atom}^0(C_x)$

and $v \notin atom^0(C_x)$, then both u and v are in $atom^0(\vec{C})$, and thus $(u, v) \in gaifman^0(\vec{C}) \subseteq gaifman(\vec{C})$. This proves Part 2(iii). This finishes the case when $e(\vec{X})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in X_0\}$,

The second interesting case is when $e(\vec{X})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in \{e_2(\vec{X})\}\}$. Let $\vec{C}$ have the types of $\vec{X}$ and let $e_2[\vec{C}/\vec{X}] \Downarrow C''$. By the induction hypothesis of either Part 1 or 2 (it does not matter which), we get $atom^0(C'') \subseteq atom^0(\vec{C})$; thus, $atom^0(C'', \vec{C}) = atom^0(\vec{C})$. Since $\{e_2(\vec{X})\}$ has flat relation type, $e_2(\vec{X})$ must have a type of the form $b \times \cdots \times b$. This means $atom^1(C'') = \{\} \subseteq atom^1(\vec{C})$; thus, $atom^1(C'', \vec{C}) = atom^1(\vec{C})$. Crucially, $atom^0(C'') \subseteq atom^0(\vec{C})$ and $atom^1(C'') = \{\}$ implies $gaifman(C'', \vec{C}) = gaifman(\vec{C})$. As C'' has no elist, $molecule^0(C'', \vec{C}) = molecule^0(\vec{C})$. Then both Part 1 and 2 of the lemma follows immediately for this case.

The other cases are straightfoward and are omitted. This completes the proof of this lemma. □

4 INTENSIONAL EXPRESSIVENESS GAP

As mentioned earlier, an intensional expressiveness gap of comprehension syntax relative to relational database systems appears to manifest in joins of low selectivity. And judging by Example 2.1, it also potentially manifests in relational intersection and relational difference, as these two operations have $O(n \log n)$ time complexity in a relational database system whereas their comprehension-syntax equivalent in Example 2.1 is quadratic. The other relational query operations (project, select, and union), as well as joins of high selectivity are succinctly expressible in $\mathcal{NRC}_1(\leq)$ with comparable time complexity when there are no indices available on the input relations; cf. Example 2.1. The relational division is ignored here because it is not directly supported by typical relational database systems; i.e., when it is needed in a relational database system, it is expressed using the other operators, usually at quadratic space and time complexity [12].

This intensional expressiveness gap is illustrated and confirmed here using two example queries on objects in canonical form. The first query, $head(x, X)$, produces the first element in an input canonical elist X, assuming this first element has the form (x, x') and x does not appear in subsequent elements of X. The second query, $zip(X, Y)$, produces an elist that pairs the ith elements in two input canonical elists X and Y of equal length, assuming the ith element of X has the form (o_i, x'_i) and that in Y has the form (o_i, y'_i) and that each o_i occurs only once in X and once in Y. These two queries are chosen because *head* can be straightforwardly implemented in constant time in any programming language, while *zip* is a very low-selectivity join which can be answered efficiently—i.e. with linear or near-linear time complexity—in relational database systems.

The expression $head'(x, X) =_{df} \{(y, y') \mid (y, y') \in X, y = x\}$ in $\mathcal{NRC}_1(\leq)$ defines the same function as *head* on any input (x, X) meeting the requirement of *head*. However, $time(head'(x, X)) = \Theta(|X|)$; i.e., it has linear time complexity.

The expression $zip'(X, Y) =_{df} \{(x, y) \mid (u, x) \in X, (v, y) \in Y, u = v\}$ in $\mathcal{NRC}_1(\leq)$ defines the same function as *zip* on any input (X, Y) meeting the requirement of *zip*. However, $time(zip'(X, Y)) = \Theta(|X| * |Y|)$; i.e., it has quadratic time complexity.

In fact, as shown below, every expression in $\mathcal{NRC}_1(\leq)$ that implements *head* has at least linear time complexity; and every expression in $\mathcal{NRC}_1(\leq)$ that implements *zip* has at least quadratic time complexity. In other words, the intensional expressiveness gap of $\mathcal{NRC}_1(\leq)$, and thus of comprehension syntax, is real.

Proposition 4.1. *Let $head(x, X) : \{b_1 \times b_2\}$ be an expression in $\mathcal{NRC}_1(\leq)$. Suppose for every object c of type b_1 and non-empty canonical object C of type $\{b_1 \times b_2\}$ whose first element is (c, c_0), and c does not appear in subsequent elements of C, $head[c/x, C/X] \Downarrow \{(c, c_0)\}$. Then $time(head[c/x, C/X] \Downarrow)$ is at least $|C|$. That is, $time(head(x, X)) = \Omega(|X|)$.*

Proof. For a contradiction, suppose $head(x, X)$ has sublinear time complexity. Then Proposition 2.5 implies $head(x, X)$ has constant time complexity. Let $head[c/x,\ C/X] \Downarrow C'$ where $C' = \{(c, c_0)\}$. As C' has type $\{b_1 \times b_2\}$, $molecule^0(C') = \{C'\}$. Similarly, $molecule^0(c, C) = \{C\}$. By Part 1 of Lemma 3.1, either $C \subseteq C'$ or $atom^1(C') \subseteq atom^0(C)$. However, $C \not\subseteq C' = \{(c, c_0)\}$ in general and $atom^1(C') = \{c, c_0\} \not\subseteq atom^0(c, C) = \{c\}$. This contradiction implies that $head(x, X)$ has at least linear time complexity. □

Proposition 4.2. *Let $zip(X, Y) : \{b_1 \times b_2\}$ be an expression in $\mathcal{NRC}_1(\leq)$ where X is a variable of type $\{b_3 \times b_1\}$, Y is a variable of type $\{b_3 \times b_2\}$, and b_1, b_2, and b_3 are distinct base types. Suppose for every canonical objects $U == \{(o_1, u_1), ..., (o_n, u_n)\}$ of type $\{b_3 \times b_1\}$ and $V == \{(o_1, v_1), ..., (o_n, v_n)\}$ of type $\{b_3 \times b_2\}$, $zip[U/X, V/Y] \Downarrow C'$ where $C' == \{(u_1, v_1), ..., (u_n, v_n)\}$. Then $time(zip[U/X, V/Y] \Downarrow)$ is at least $|U| * |V|$. Thus, $time(zip(X, Y)) = \Omega(|U| * |Y|)$.*

Proof. Suppose for a contradiction that $zip(X, Y)$ has subquadratic time complexity. Then Proposition 2.5 implies $zip(X, Y)$ has either constant or linear time complexity.

Assume $zip(X, Y)$ has constant time complexity and $zip[U/X, V/Y] \Downarrow C'$ where $C' == \{(u_1, v_1),$ $..., (u_n, v_n)\}$. Clearly, $molecule^0(C') = \{C'\}$ and $molecule^0(U, V) = \{U, V\}$. Then, by Part 1 of Lemma 3.1, either $U \subseteq C'$, $V \subseteq C'$, or $atom^1(C') \subseteq atom^0(U, V) = \{\}$. Clearly, all three options are impossible. Thus $zip(X, Y)$ cannot have constant time complexity.

Suppose instead $zip(X, Y)$ has linear time complexity. Then $gaifman(C') = C' = \{(u_1, v_1), ...,$ $(u_n, v_n)\}$. However, for $1 \leq i \leq n$, $(u_i, v_i) \in gaifman(C') \notin gaifman(U, V) = U \cup V$. Then, by Part 2 of Lemma 3.1, either $u_i \in atom^0(U, V)$ or $v_i \in atom^0(U, V)$. However, as U and V are both elists, $atom^0(U, V) = \{\}$ and thus contains neither u_i nor v_i. Thus $zip(X, Y)$ cannot have linear time complexity. Therefore, it has at least quadratic time complexity. □

5 NON-SOLUTIONS

As $\mathcal{NRC}_1(\leq)$ is unable to express some relational queries efficiently, it is pertinent to augment it with new constructs to enable new algorithms for the sake of practicality. It is also pertinent to investigate and to precisely characterize its intensional expressiveness gap. For both of these purposes, it is worth considering functions in typical programming language libraries as candidates. Four functions which are commonly found in the collection-type libraries of modern programming languages stand out: *takewhile*, *dropwhile*, *fold*, and *zip*.

The function $takewhile(p)(X)$ iterates on the list X as long as the predicate p is true on the current element in X, returning this element and moves on to the next element; the iteration stops as soon as p becomes false. The function $dropwhile(p)(X)$ iterates on the list X as long as the predicate p is true on the current element in X, dropping this element and moves on to the next element; the iteration stops as soon as p becomes false, returning the remaining elements of X. The time complexity of *takewhile* and *dropwhile* thus can be bounded by a constant under the situation that p is true for a small initial segment of X. This stopping-mid-way feature makes *takewhile* and *dropwhile* potential candidates for addressing the intensional expressiveness gap of $\mathcal{NRC}_1(\leq)$, as the sole iteration construct $\bigcup\{e_1 \mid x \in e_2\}$ of $\mathcal{NRC}_1(\leq)$ must access every elements of e_2 and cannot stop mid way.

The function $fold(f, x_0)(X)$ corresponds to a primitive recursive function on a list. It is called structural recursion by some in our community [4]. It iterates on the elist X, applying the function f at each step to the current element and an accumulator (which is initialized to x_0), and returns the final value of the accumulator. This function has high extensional expressive power, as it can express many queries—e.g., transitive closure [4, 20]—which are inexpressible in $\mathcal{NRC}(\leq)$. Moreover, it can express some of these queries efficiently; e.g. $\mathcal{NRC}(fold)$ can express transitive closure with

Additions from popular function libraries

$$\frac{e_1 : \{s\} \quad e_2 : \mathbb{B}}{\mathit{takewhile}(x^s \in e_1,\ e_2) : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \mathbb{B}}{\mathit{dropwhile}(x^s \in e_1,\ e_2) : \{s\}}$$

$$\frac{e_1 : s_1 \quad e_2 : \{s_2\} \quad e_3 : s}{\mathit{fold}(x^{s_1} := e_1,\ y^{s_2} \in e_2,\ e_3) : s} \qquad \frac{e : \{s\}}{\mathit{sort}(e) : \{s\}}$$

Their operational semantics

$$\frac{\begin{array}{c}\Gamma, e_1 \Downarrow \Gamma', \{C_1, \ldots, C_j, \ldots, C_n\} \\ \Gamma', e_2[C_1/x] \Downarrow \Gamma_1, \mathit{true} \\ \vdots \\ \Gamma_{j-1}, e_2[C_j/x] \Downarrow \Gamma_j, \mathit{true} \\ \Gamma_j, e_2[C_{j+1}/x] \Downarrow \Gamma_{j+1}, \mathit{false}\end{array}}{\Gamma, \mathit{takewhile}(x \in e_1,\ e_2) \Downarrow \Gamma_{j+1}, \{C_1, \ldots, C_j\}} \qquad \frac{\begin{array}{c}\Gamma, e_1 \Downarrow \Gamma', \{C_1, \ldots, C_n\} \\ \Gamma', e_2[C_1/x] \Downarrow \Gamma_1, \mathit{true} \\ \vdots \\ \Gamma_{n-1}, e_2[C_n/x] \Downarrow \Gamma_n, \mathit{true}\end{array}}{\Gamma, \mathit{takewhile}(x \in e_1,\ e_2) \Downarrow \Gamma_n, \{C_1, \ldots, C_n\}}$$

$$\frac{\begin{array}{c}\Gamma, e_1 \Downarrow \Gamma', \{C_1, \ldots, C_j, \ldots, C_n\} \\ \Gamma', e_2[C_1/x] \Downarrow \Gamma_1, \mathit{true} \\ \vdots \\ \Gamma_{j-1}, e_2[C_j/x] \Downarrow \Gamma_j, \mathit{true} \\ \Gamma_j, e_2[C_{j+1}/x] \Downarrow \Gamma_{j+1}, \mathit{false}\end{array}}{\Gamma, \mathit{dropwhile}(x \in e_1,\ e_2) \Downarrow \Gamma_{j+1}, \{C_{j+1}, \ldots, C_n\}} \qquad \frac{\begin{array}{c}\Gamma, e_1 \Downarrow \Gamma', \{C_1, \ldots, C_n\} \\ \Gamma', e_2[C_1/x] \Downarrow \Gamma_1, \mathit{true} \\ \vdots \\ \Gamma_{n-1}, e_2[C_n/x] \Downarrow \Gamma_n, \mathit{true}\end{array}}{\Gamma, \mathit{dropwhile}(x \in e_1,\ e_2) \Downarrow \Gamma_n, \{\}}$$

$$\frac{\begin{array}{c}\Gamma, e_1 \Downarrow \Gamma', C'_0 \quad \Gamma', e_2 \Downarrow \Gamma_0, \{C_1, \ldots, C_n\} \\ \Gamma_0, e_3[C'_0/x, C_1/y] \Downarrow \Gamma_1, C'_1 \\ \vdots \\ \Gamma_n, e_3[C'_{n-1}/x, C_n/y] \Downarrow \Gamma_n, C'_n\end{array}}{\Gamma, \mathit{fold}(x := e_1,\ y \in e_2,\ e_3) \Downarrow \Gamma_n, C'_n} \qquad \frac{\Gamma, e \Downarrow \Gamma', C'}{\Gamma, \mathit{sort}(e) \Downarrow \Gamma', C}\ C == \mathit{canonize}(C')$$

Fig. 4. The syntax and operational semantics of *takewhile*, *dropwhile*, and *fold*.

quadratic time complexity [20], whereas $\mathcal{NRC}$ endowed instead with a powerset operator can only express transitive closure at exponential time complexity [19, 25].

The function $\mathit{zip}(X, Y)$ iterates on X and Y in lock step, pairing elements in the two collections strictly based on their positions, viz. the ith element of X with the ith element of Y. It is arguably the most common, and quite often, the only function in the function libraries of programming languages that performs synchronized iteration on two collections.

Unfortunately, as shown in subsections below, augmenting $\mathcal{NRC}_1(\leq)$ with any of these functions does not escape the limited-mixing lemma.

5.1 Adding *takewhile* and *dropwhile*

The constructs for *takewhile* and *dropwhile*, as well as their operational semantics, are provided in Figure 4. Briefly, $takewhile(x \in e_1,\ e_2) = \{C_1, ..., C_i\}$ where $p(x) = e_2$; $e_1 == \{C_1, ..., C_i, C_{i+1}, ..., C_n\}$; and $p(C_1), ..., p(C_i)$ are all true, and $p(C_{i+1})$ is false if C_{i+1} exists. Whereas, $dropwhile(x \in e_1, e_2) = \{C_{i+1}, ..., C_n\}$ where $p(x) = e_2$; $e_1 == \{C_1, ..., C_i, C_{i+1}, ..., C_n\}$; and $p(C_1), ..., p(C_i)$ are all true, and $p(C_{i+1})$ is false if C_{i+1} exists. Note that the notation $x \in e_1$ in $takewhile(x \in e_1,\ e_2)$ and $dropwhile(x \in e_1,\ e_2)$ introduces a fresh variable x whose scope is the expression e_2 in these constructs; i.e. it is not a set membership test.

The definitions of *takewhile* and *dropwhile* require access to the elist representation of e_1. Recall that two distinct elists $U == \{U_1, ..., U_n\}$ and $V == \{V_1, ..., V_m\}$ can represent the same set. When $U = V$, it is possible that $takewhile(x \in U,\ e_2) \neq takewhile(x \in V,\ e_2)$ and $dropwhile(x \in U,\ e_2) \neq dropwhile(x \in V,\ e_2)$. E.g., suppose $e_2(U_1)$ is true and $e_2(U_2)$ is false, and $U == \{U_1, U_2\}$ and $V == \{U_2, U_1\}$; then $U = V$ but $takewhile(x \in U,\ e_2) == \{U_1\} \neq takewhile(x \in V,\ e_2) == \{\}$.

To deal with such a situation, a restriction must be imposed on the use of *takewhile* and *dropwhile* to ensure soundness. Specifically, the subexpression e_1 in $takewhile(x \in e_1,\ e_2)$ and $dropwhile(x \in e_1,\ e_2)$ must evaluate to an elist in canonical form at runtime. Thus, an additional contruct $sort(e)$ is provided for explicit conversion of any object e to its canonical form when there is need to do so.

In practice, $sort(X)$ has $O(|X| \log(|X|))$ time complexity on flat relations. But in the operational semantics, it is made to return the canonical form instantaneously; i.e., it has constant time complexity according to the operational semantics. Making *sort* instantaneous is a matter of convenience. The results in this paper are about lower bounds, which can only become worse if *sort* is not instantaneous. Indeed, the results below show that $\mathcal{NRC}_1(\leq, takewhile, dropwhile, sort)$ retains a form of the limited-mixing property, and thus still cannot express *zip* efficiently despite the availability of instantaneous sorting.

Lemma 5.1 (Limited mixing). *Let $e(\vec{X}) : s$ be an expression in $\mathcal{NRC}_1(takewhile, dropwhile, sort)$. Suppose objects $\vec{C}$ have the same types as $\vec{X}$, and $e(\vec{C}/\vec{X}) \Downarrow C'$.*

(1) *If $e(\vec{X})$ has constant time complexity, then there is a number k that depends only on $e(\vec{X})$ but not on $\vec{C}$, and a set $A \subseteq atom^{\leq 1}(\vec{C})$ where $|A| \leq k$, and*
(i) $atom^0(C') \subseteq atom^0(\vec{C})$,
(ii) $atom^1(C') \subseteq atom^{\leq 1}(\vec{C})$,
(iii) for each $(u, v) \in gaifman(\vec{C}')$, either $(u, v) \in gaifman(\vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in A$, or $v \in atom^0(\vec{C})$ and $u \in A$, or $u \in A$ and $v \in A$.

(2) *If $e(\vec{X})$ has linear time complexity, then there is a number k that depends only on $e(\vec{X})$ but not on $\vec{C}$, and a set $A \subseteq atom^{\leq 1}(\vec{C})$ where $|A| \leq k$, and*
(i) $atom^0(C') \subseteq atom^0(\vec{C})$,
(ii) $atom^1(C') \subseteq atom^{\leq 1}(\vec{C})$, and
(iii) for each $(u, v) \in gaifman(C')$, either $(u, v) \in gaifman(\vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $u \in atom^1(\vec{C})$ and $v \in atom^0(\vec{C})$, or $u \in A$ and $v \in atom^1(\vec{C})$, or $v \in A$ and $u \in atom^1(\vec{C})$.

Proof. Without loss of generality, $e(\vec{X})$ is assumed to be in the normal form induced by the restricted system of rewrite rules. The proof proceeds by induction based on an ordering $\sqsubseteq$ on the structural complexity of expressions, defined as the smallest reflexive transitive relation having the following properties: (i) $e' \sqsubseteq e$ if e' is a subexpression of e; (ii) $e' \sqsubseteq e$ if there is $e'' \sqsubseteq e$ and e' is an expression syntactically identical to e'' after replacing a subexpression of e'' by a fresh variable Y having the same type as the replaced subexpression.

The first interesting case is when $e(\vec{X})$ has the form $\mathit{takewhile}(x \in e_1(\vec{X}),\ e_2(x, \vec{X}))$. Let $e_1[\vec{C}/\vec{X}] \Downarrow C''$ and $\mathit{takewhile}(x \in e_1[\vec{C}/\vec{X}],\ e_2[\vec{C}/\vec{X}](x)) \Downarrow C'$. The induction hypotheses of Part 1 and 2 are applicable to $e_1(\vec{X})$. Thus, C'' has the properties of Part 1(i) to (iii) and Part 2(i) to (iii). By definition of *takewhile*, $C' \subseteq C''$. Thus, C' has the properties of Part 1(i) to (iii) and Part 2(i) to (iii). This finishes the case when $e(\vec{X})$ has the form $\mathit{takewhile}(x \in e_1(\vec{X}),\ e_2(x, \vec{X}))$.

The second interesting case is when $e(\vec{X})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in \mathit{takewhile}(y \in e_2(\vec{X}), e_3(y, \vec{X}))\}$. Let $\mathit{takewhile}(y \in e_2[\vec{C}/\vec{X}], e_3[\vec{C}/\vec{X}](y)) \Downarrow C''$ and $\bigcup\{e_1[\vec{C}/\vec{X}](x) \mid x \in Y[C''/Y]\} \Downarrow C'$. For Part 1, the induction hypothesis on the *takewhile* subexpression implies C'' has the properties of Part 1(i) to (iii): $\mathit{atom}^0(C'') \subseteq \mathit{atom}^0(\vec{C})$; $\mathit{atom}^1(C'') \subseteq \mathit{atom}^{\leq 1}(\vec{C})$; and for each $(u, v) \in \mathit{gaifman}(C'')$, either $(u, v) \in \mathit{gaifman}(\vec{C})$, or $u \in \mathit{atom}^0(\vec{C})$ and $v \in A''$, or $v \in \mathit{atom}^0(\vec{C})$ and $u \in A''$, or $u \in A''$ and $v \in A''$, where $A'' \subseteq \mathit{atom}^{\leq 1}(\vec{C})$ and $|A''| \leq k$. Here, k'' is a number that depends only on $\mathit{takewhile}(y \in e_2(\vec{X}), e_3(y, \vec{X}))\}$ and not on $\vec{C}$.

Since $\bigcup\{e_1(x, \vec{X}) \mid x \in Y\} \sqsubseteq e(\vec{X})$, the induction hypothesis is applicable to it. So, C' has the properties of Part 1(i) to (iii): $\mathit{atom}^0(C') \subseteq \mathit{atom}^0(C'', \vec{C}) = \mathit{atom}^0(\vec{C})$; $\mathit{atom}^1(C') \subseteq \mathit{atom}^{\leq 1}(C'', \vec{C}) = A'' \cup\ \mathit{atom}^{\leq 1}(\vec{C}) = \mathit{atom}^{\leq 1}(\vec{C})$; and for each $(u, v) \in \mathit{gaifman}(C')$, either $(u, v) \in \mathit{gaifman}(C'', \vec{C})$, or $u \in \mathit{atom}^0(C'', \vec{C}) = \mathit{atom}^0(\vec{C})$ and $v \in A'$, or $v \in \mathit{atom}^0(C'', \vec{C}) = \mathit{atom}^0(\vec{C})$ and $u \in A'$, or $u \in A'$ and $v \in A'$, where $A' \subseteq \mathit{atom}^{\leq 1}(C'', \vec{C})$ and $|A'| \leq k'$. Here, k' is a number that depends only on $\bigcup\{e_1(x, \vec{X}) \mid x \in Y\}$ and not on C'' and $\vec{C}$.

Let $A = A' \cup\ A''$ and $k = k' + k''$. Then Part 1(i) to (iii) are trivially satisfied. This completes the proof of Part 1 for the case when $e(\vec{C})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in \mathit{takewhile}(y \in e_2(\vec{X}), e_3(y, \vec{X}))\}$. For Part 2, the argument is similar to Part 1 and is omitted.

The two cases for *dropwhile* are similar to the cases for *takewhile*. The case for *sort* is trivial. The other cases are similar to the proof of Lemma 3.1. All these cases are thus omitted. □

Corollary 5.2. *Let $zip(X, Y) : \{b_1 \times b_2\}$ be an expression in $\mathcal{NRC}_1(\mathit{takewhile}, \mathit{dropwhile}, \mathit{sort})$ where X is a variable of type $\{b_3 \times b_1\}$, Y is a variable of type $\{b_3 \times b_2\}$, and b_1, b_2, and b_3 are distinct base types. Suppose for every canonical objects $U == \{(o_1, u_1), ..., (o_n, u_n)\}$ of type $\{b_3 \times b_1\}$ and $V == \{(o_1, v_1), ..., (o_n, v_n)\}$ of type $\{b_3 \times b_2\}$, $zip[U/X, V/Y] \Downarrow C'$ where $C' == \{(u_1, v_1), ..., (u_n, v_n)\}$. Then $\mathit{time}(zip[U/X, V/Y] \Downarrow)$ is at least $|U| * |V|$.*

Proof. Suppose $zip(X, Y)$ has constant or linear time complexity and $zip[U/X, V/Y] \Downarrow C'$ where $C' == \{(u_1, v_1), ..., (u_n, v_n)\}$. Since U and V are sets, $\mathit{atom}^0(U, V) = \{\}$. By Part 1 of Lemma 5.1, for each $(u_i, v_i) \in C'$, it is clear that $(u_i, v_i) \notin U$. So either $u_i \in \mathit{atom}^0(U, V)$, or $v_i \in \mathit{atom}^0(U, V)$, or $u_i \in A$, or $v_i \in A$. As $\mathit{atom}^0(U, V) = \{\}$, A has to contain every u_i or v_i. So, $|A| = n$ cannot be independent of (the size of) U and V. Hence, $zip(X, Y)$ cannot have constant or linear time complexity in $\mathcal{NRC}_1(\mathit{takewhile}, \mathit{dropwhile}, \mathit{sort})$. Then, by Proposition 2.5, it must have at least quadratic time complexity. □

Notice that Part 1 of Lemma 5.1 is weaker than Part 1 of Lemma 3.1. The latter says that for any constant-time function of $\mathcal{NRC}_1(\leq)$, its output either does not contain any part of the input flat relations or it must contain whole-copy of them. Whereas, a constant-time function of $\mathcal{NRC}_1(\leq, \mathit{takewhile}, \mathit{dropwhile}, \mathit{sort})$ can return a subset of an input flat relation. In particular, $\mathit{head}(x, X) =_{df} \mathit{takewhile}(y \in X, x = y.\pi_1)$ defines a *head* function which has constant time complexity on inputs meeting the requirements for *head* specified earlier. So, *takewhile* and *dropwhile* partially address the intensional expressiveness gap of comprehension syntax.

It is worth further remarking that, as a function, both *takewhile* and *dropwhile* are expressible in $\mathcal{NRC}_1(\leq)$. I.e., they do not contribute to the extensional expressive power of $\mathcal{NRC}_1(\leq)$; rather, they contribute strictly to the intensional expressive power of $\mathcal{NRC}_1(\leq)$, albeit still falling short of filling its intensional expressiveness gap.

Proposition 5.3. *Every function expressible in $\mathcal{NRC}_1(\leq$, takewhile, dropwhile, sort$)$ is already expressible in $\mathcal{NRC}_1(\leq)$. That is, the former is a conservative extension of the latter, even though more algorithms are expressible using the former.*

Proof. Suppose $p(x^s) : \mathbb{B}$ and $R : \{s\}$ is canonical. Let $sentinels =_{df} \{y \mid y \in R, p(y) = false\}$, and $first =_{df} \{y \mid y \in sentinels, \{z \mid z \in sentinels, z < y\}\ isempty\}$. Then, $takewhile(x \in R,\ p(x)) =$ *if first isempty then R else* $\{x \mid y \in first, x \in R, x < y\}$. And, $dropwhile(x \in R, p(x)) = \{x \mid y \in first, x \in R, y \leq x\}$. Therefore, *takewhile* and *dropwhile* are both definable as functions in $\mathcal{NRC}_1(\leq)$. □

A reader from the programming language community might question the lack of head and tail expression constructs in the $\mathcal{NRC}$ variants considered so far. This is because they are quite useless for the purpose of filling the intensional expressiveness gap of comprehension syntax. Let $take_n(X)$ returns the first n elements of a set X in canonical form in constant time; e.g. $take_1(X)$ corresponds to the "head" of a set. Let $drop_n(X)$ drops the first n elements of a set X in canonical form in constant time; e.g. $drop_1(X)$ corresponds to the "tail" of a set. It is easy to see that an analog of Lemma 5.1 can be proved for $\mathcal{NRC}_1(\leq$, $take_n$, $drop_n$, $sort)$ by copying the proof of Lemma 5.1 more or less word for word.

5.2 Adding *fold*

The construct for *fold* and its operational semantics are provided in Figure 4. Briefly, $fold(x := e_1, y \in e_2, e_3) = f(\ldots(f(f(C_0, C_1), C_2), \ldots), C_n)$, where $f(x, y) = e_3$, $e_1 = C_0$, and $e_2 == \{C_1, ..., C_n\}$. Note that the notations $x := e_1$ and $y \in e_2$ in $fold(x := e_1,\ y \in e_2,\ e_3)$ introduce fresh variables x and y, whose scope is the expression e_3.

The definition of *fold* requires access to the elist representation of e_2. To ensure soundness, as in the case for *takewhile* and *dropwhile*, a restriction is imposed that e_2 must evaluate to an elist in canonical form. Again, this is facilitated by explicitly using *sort* to convert objects to their canonical form when needed.

In contrast to *takewhile* and *dropwhile*, which form a conservative extension of $\mathcal{NRC}_1(\leq)$, *fold* adds considerable extensional expressive power to $\mathcal{NRC}_1(\leq)$ [20]. Yet, in spite of it high extensional expressive power, it does not address the intensional expressiveness gap.

Lemma 5.4 (Limited mixing). *Let $e(\vec{X}) : s$ be an expression in $\mathcal{NRC}_1(\leq, fold, sort)$. Suppose objects $\vec{C}$ have the same types as $\vec{X}$, and $e(\vec{C}/\vec{X}) \Downarrow C'$.*

(1) *If $e(\vec{X})$ has constant time complexity, then*
 (i) $atom^0(C') \subseteq atom^0(\vec{C})$,
 (ii) $atom^1(C') \subseteq atom^{\leq 1}(\vec{C})$,
 (iii) $gaifman(C') \subseteq gaifman(\vec{C})$, and
 (iv) for each $U \in molecule^0(C')$, there are V_0, V_1, ..., V_m such that $atom^1(V_0) \subseteq atom^0(\vec{C})$, $V_j \in molecule^0(\vec{C})$ for each $1 \leq j \leq m$, and $U = V_0 \cup V_1 \cup \cdots \cup V_m$.

(2) *If $e(\vec{X})$ has linear time complexity, then there is a number k that depends only on $e(\vec{X})$ but not on $\vec{C}$, and a set $A \subseteq atom^{\leq 1}(\vec{C})$ where $|A| \leq k$, and*
 (i) $atom^0(C') \subseteq atom^0(\vec{C}) \cup A$,
 (ii) $atom^1(C') \subseteq atom^{\leq 1}(\vec{C})$, and
 (iii) for each $(u, v) \in gaifman(C')$, either $(u, v) \in gaifman(\vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $u \in atom^1(\vec{C})$ and $v \in atom^0(\vec{C})$, or $u \in A$ and $v \in atom^1(\vec{C})$, or $v \in A$ and $u \in atom^1(\vec{C})$.

Proof. Without loss of generality, $e(\vec{X})$ is assumed to be in the normal form induced by the restricted system of rewrite rules. The proof proceeds by induction on the structural complexity of expressions; i.e. the ordering $\sqsubseteq$ on expressions. As the proof is long, it is divided into subsections for better readability.

***

The first case is when $e(\vec{X})$ has the form $fold(x := e_1(\vec{X}),\ y \in e_2(\vec{X}),\ e_3(x, y, \vec{X}))$. For Part 1, the entire expression is assumed to have constant time complexity. So, the induction hypothesis of Part 1 is applicable to subexpressions $e_1(\vec{X})$, $e_2(\vec{X})$, and $e_3(x, y, \vec{X})$. Suppose $e_1[\vec{C}/\vec{X}] \Downarrow C'_0$, $e_2[\vec{C}/\vec{X}] \Downarrow C''$ where $C'' == \{C_1, ..., C_n\}$ and $n < h$ for some number h which depends only on $e_2(\vec{X})$ but not on $\vec{C}$. Note that if h depends on $\vec{C}$, the complexity of the whole *fold* expression can no longer be constant time, rendering the case vacuously true. This also implies $atom^1(C'') \subseteq atom^0(\vec{C})$, and $atom^0(C_j) \subseteq atom^0(\vec{C})$ for $1 \leq j \leq n$; otherwise, there is a $V \in molecule^0(\vec{C})$ such that $V \subseteq C''$, which implies $n \geq |V|$ and thus the whole *fold* expression cannot be constant time.

Now, let $e_3[C'_0/x, C_1/y, \vec{C}/\vec{X}] \Downarrow C'_1$, ..., $e_3[C'_{n-1}/x, C_n/y, \vec{C}/\vec{X}] \Downarrow C'_n$. By the induction hypothesis of Part 1 on $e_2[\vec{C}/\vec{X}]$, we know Part 1 holds on $C'' == \{C_1, ..., C_n\}$. Also, each $C_j \in C''$ has type of the form $b \times \cdots \times b$ because we only have flat relations. Thus, for each $C_j \in C''$, we have $atom^0(C_j) \subseteq atom^0(\vec{C})$, $atom^1(C_j) = \{\}$, and $gaifman(C_j) \subseteq gaifman(\vec{C})$. By the induction hypothesis of Part 1 on $e_1(\vec{X})$, we know $atom^0(C'_0) \subseteq atom^0(\vec{C})$, $atom^1(C'_0) \subseteq atom^{\leq 1}(\vec{C})$, and $gaifman(C'_0) \subseteq gaifman(\vec{C})$. This means $atom^0(C_1, C'_0, \vec{C}) \subseteq atom^0(\vec{C})$, $atom^1(C_1, C'_0, \vec{C}) \subseteq atom^{\leq 1}(\vec{C})$, and $gaifman(C_1, C'_0, \vec{C}) \subseteq gaifman(\vec{C})$. By the induction hypothesis of Part 1 on $e_3(x, y, \vec{X})$, $atom^0(C'_1) \subseteq atom^0(C_1, C'_0, \vec{C}) \subseteq atom^0(\vec{C})$, $atom^1(C'_1) \subseteq atom^{\leq 1}(C_1, C'_0, \vec{C}) \subseteq atom^{\leq 1}(\vec{C})$, and $gaifman(C'_1) \subseteq gaifman(C_1, C'_0, \vec{C}) \subseteq gaifman(\vec{C})$. Now, by an induction on $1 \leq j < n$, we get $atom^0(C'_{j+1}) \subseteq atom^0(C_{j+1}, C'_j, \vec{C}) \subseteq atom^0(\vec{C})$, $atom^1(C'_{j+1}) \subseteq atom^{\leq 1}(C_{j+1}, C'_j, \vec{C}) \subseteq atom^{\leq 1}(\vec{C})$, and $gaifman(C'_{j+1}) \subseteq gaifman(C_{j+1}, C'_j, \vec{C}) \subseteq gaifman(\vec{C})$. This settles Part 1(i)-(iii) for this case.

By the induction hypothesis of Part 1 on $e_2(\vec{X})$, there are $V_{0,0}, V_{0,1}, ..., V_{0,m_0}$ such that $atom^1(V_{0,0}) \subseteq atom^0(\vec{C})$, $V_{0,j} \in molecule^0(\vec{C})$ for $1 \leq j \leq m_0$, and $C'_0 = V_{0,0} \cup V_{0,1} \cup \cdots \cup V_{0,m}$. By the induction hypothesis of Part 1 on $e_1(x, y, \vec{X})$, there are $V_{1,0}, V_{1,1}, ..., V_{1,m_1}$ such that $atom^1(V_{1,0}) \subseteq atom^0(C_1, C'_0, \vec{C}) \subseteq atom^0(\vec{C})$, $V_{1,j} \in molecule^0(C'_0, \vec{C})$ for $1 \leq j \leq m_0$, and $C'_1 = V_{1,0} \cup V_{1,1} \cup \cdots \cup V_{1,m_1}$. Since $C'_0 = V_{0,0} \cup V_{0,1} \cup \cdots \cup V_{0,m}$, if any of $V_{1,j}$ is C'_0, we just replace $V_{1,j}$ above with $V_{0,1} \cup \cdots \cup V_{0,m}$ and $V_{1,0}$ above with $V_{1,0} \cup V_{0,0}$, which establishes Part 1(iv) for C'_1. Now, by an induction on $1 \leq j < n$, we get Part 1(iv) on each C'_j. This proves Part 1(iv) for this case.

***

For Part 2, the expression $fold(x := e_1(\vec{X}),\ y \in e_2(\vec{X}),\ e_3(x, y, \vec{X}))$ is assumed to have linear time complexity. There are two subcases. The first subcase is $e_2(\vec{X})$ has linear time complexity; but $e_2[\vec{C}/\vec{X}] \Downarrow C''$, $C'' == \{C_1, ..., C_n\}$ and $n < h$ for some h which depends on $e_2(\vec{X})$ but not on $\vec{C}$. Then Part 2 of the induction hypothesis is applicable to $e_1(\vec{X})$, $e_2(\vec{X})$, and $e_3(x, y, \vec{X})$. Suppose $e_1[\vec{C}/\vec{X}] \Downarrow C'_0$, $e_3[C'_0/x, C_1/y, \vec{C}/\vec{X}] \Downarrow C'_1$, ..., $e_3[C'_{n-1}/x, C_n/y, \vec{C}/\vec{X}] \Downarrow C'_n$. Let k'' and A'' denote the number k and the set A induced by Part 2 from $e_2[\vec{C}/\vec{X}]$. Let k_0 and A_0 denote the number k and the set A induced by Part 2 from $e_1[\vec{C}/\vec{X}]$. Let k_j and A'_j denote the number k and the set A induced by Part 2 from $e_3[C_{j-1}/x, C_j/y, \vec{C}/\vec{X}]$, and $A_j = A'_j \cup atom^0(C_j)$, for $1 \leq j \leq n$. Note that an upperbound l on the number of level-0 atoms in C_j is determined by the type of C_j

(i.e. the type of y). Hence, we can set $k = k'' + k_0 + k_1 + \cdots + k_n + n * l$ as the k for Part 2 for the whole expression, and $A = A'' \cup A_0 \cup \cdots \cup A_n$ as the A for Part 2 for the whole expression.

By the induction hypothesis of Part 2(ii) on $e_1(\vec{X})$, we get $atom^1(C'_0) \subseteq atom^{\leq 1}(\vec{C})$. And on $e_2(\vec{X})$, we get $atom^1(C'') \subseteq atom^{\leq 1}(\vec{C})$; this implies $atom^{\leq 1}(C_j) = atom^0(C_j) \subseteq atom^{\leq 1}(\vec{C})$, for $1 \leq j \leq n$. Finally, on $e_3(x, y, \vec{X})$, we get $atom^1(C'_j) \subseteq atom^{\leq 1}(C'_{j-1}, C_j, \vec{C}) = atom^{\leq 1}(C'_{j-1}) \cup atom^{\leq 1}(C_j) \cup atom^{\leq 1}(\vec{C}) \subseteq atom^{\leq 1}(\vec{C})$. In particular, $atom^1(C'_n) \subseteq atom^{\leq 1}(\vec{C})$, proving Part 2(ii) for the whole expression.

By the induction hypothesis of Part 2(i) on $e_1(\vec{X})$, we get $atom^0(C'_0) \subseteq atom^0(\vec{C}) \cup A'_0 \subseteq atom^0(\vec{C}) \cup A$. And on $e_2(\vec{X})$, we get $atom^0(C'') = \{\} \subseteq atom^0(\vec{C}) \cup A'' \subseteq atom^0(\vec{C}) \cup A$. Finally, on $e_3(x, y, \vec{X})$, we get $atom^0(C'_j) \subseteq atom^0(C'_{j-1}, C_j, \vec{C}) = atom^0(C'_{j-1}) \cup atom^0(C_j) \cup atom^0(\vec{C}) \subseteq atom^0(\vec{C}) \cup A$, for $1 \leq j \leq n$. In particular, $atom^0(C'_n) \subseteq atom^0(\vec{C}) \cup A$, proving Part 2(i) for the whole expression.

By the induction hypothesis of Part 2(iii) on $e_1(\vec{X})$, we get for each $(u, v) \in gaifman(C'_0)$, either $(u, v) \in gaifman(\vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $u \in atom^1(\vec{C})$ and $v \in atom^0(\vec{C})$, or $u \in A'_0 \subseteq A$ and $v \in atom^1(\vec{C})$, or $u \in atom^1(\vec{C})$ and $v \in A'_0 \subseteq A$. And on $e_2(\vec{X})$, we get for each $(u, v) \in gaifman(C'')$, either $C_j = (u, v) \in gaifman(\vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $u \in atom^1(\vec{C})$ and $v \in atom^0(\vec{C})$, or $u \in A'' \subseteq A$ and $v \in atom^1(\vec{C})$, or $u \in atom^1(\vec{C})$ and $v \in A'' \subseteq A$. Finally, on $e_3(x, y, \vec{X})$, we get for each $(u, v) \in gaifman(C'_j)$, either $(u, v) \in gaifman(C'_{j-1}, C_j, \vec{C})$, or $u \in atom^0(C'_{j-1}, C_j, \vec{C})$ and $v \in atom^1(C'_{j-1}, C_j, \vec{C})$, or $u \in atom^1(C'_{j-1}, C_j, \vec{C})$ and $v \in atom^0(C'_{j-1}, C_j, \vec{C})$, or $u \in A'' \subseteq A$ and $v \in atom^1(C'_{j-1}, C_j, \vec{C})$, or $u \in atom^1(C'_{j-1}, C_j, \vec{C})$ and $v \in A'' \subseteq A$.

Note that $atom^1(C'_{j-1}, C_j, \vec{C}) = atom^1(C'_{j-1}) \cup atom^1(C_j) \cup atom^1(\vec{C}) = atom^{\leq 1}(\vec{C}) = atom^0(\vec{C}) \cup atom^1(\vec{C})$, while $atom^0(C'_{j-1}, C_j, \vec{C}) = atom^0(C'_{j-1}) \cup atom^0(C_j) \cup atom^0(\vec{C}) \subseteq atom^0(\vec{C}) \cup A$. Note also that $u, v \in atom^0(\vec{C})$ implies $(u, v) \in gaifman(\vec{C})$. Therefore, after removing some redundancies, we get for each $(u, v) \in gaifman(C'_j)$, either $(u, v) \in gaifman(C'_{j-1}, C_j, \vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $v \in atom^0(\vec{C})$ and $u \in atom^1(\vec{C})$, or $u \in A$ and $v \in atom^1(\vec{C})$, or $v \in A$ and $u \in atom^1(\vec{C})$. Finally, an induction on j gets us for each $(u, v) \in gaifman(C'_j)$, either $(u, v) \in gaifman(C'_{j-1}, C_j, \vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $v \in atom^0(\vec{C})$ and $u \in atom^1(\vec{C})$, or $u \in A$ and $v \in atom^1(\vec{C})$, or $v \in A$ and $u \in atom^1(\vec{C})$. This proves Part 2(iii) for the whole expression, as well as settling the subcase.

***

The second subcase, when $fold(x := e_1(\vec{X}), y \in e_2(\vec{X}), e_3(x, y, \vec{X}))$ **has linear time complexity, is when** $e_2(\vec{X})$ **has linear time complexity,** $e_2[\vec{C}/\vec{X}] \Downarrow C''$**, and** $C'' == \{C_1, ..., C_n\}$ **where** n **depends on** $\vec{C}$**.** Then $e_3(x, y, \vec{X})$ has constant time complexity; otherwise, the whole expression gets quadratic time complexity.

Let k'' and A'' denote the number k and the set A induced by Part 2 from $e_2[\vec{C}/\vec{X}] \Downarrow C''$. Let k_0 and A_0 denote the number k and the set A induced by Part 2 from $e_1[\vec{C}/\vec{X}] \Downarrow C'_0$. Furthermore, without loss of generality, let the type of $e_3(x, y, \vec{X})$ be $s_1 \times \cdots \times s_h$, where s_i is either a base type b or a set type. Let k' be the number of s_i that is a base type. Then we can set $k = k' + k'' + k_0$ as the k. Let A' be a set of up to k' level-0 or level-1 atoms of $\vec{C}$ to be chosen later. Then we set $A = A' \cup A'' \cup A_0$ as the A for Part 2 of the whole expression.

The number of level-0 atoms in C'_n is actually upperbounded by the type of $e_3(x, y, \vec{X})$; specifically, this upperbound is k'. Thus, by setting A' to the level-0 atoms in C'_n, we get $atom^0(C'_n) = A' \subseteq A \subseteq atom^0(\vec{C}) \cup A$, trivially proving Part 2(i) for the whole expression.

The induction hypothesis of Part 2(ii) on $e_1(\vec{X})$ gives $atom^1(C'_0) \subseteq atom^{\leq 1}(\vec{C})$. The induction hypothesis of Part 2(ii) on $e_2(\vec{X})$ gives $atom^1(C'') \subseteq atom^{\leq 1}(\vec{C})$, implying $atom^0(C_j) \subseteq atom^{\leq 1}(\vec{C})$. Since $e_3(x, y, \vec{X})$ has constant time complexity, the induction hypothesis of Part 2(ii) gives $atom^1(C'_j) \subseteq atom^{\leq 1}(C'_{j-1}, C_j, \vec{C}) = atom^{\leq 1}(C'_{j-1}) \cup atom^{\leq C_j} \cup atom^{\leq 1}(\vec{C}) = atom^{\leq 1}(\vec{C})$, for $1 \leq j \leq n$. In particular, $atom^1(C'_n) \subseteq atom^{\leq 1}(\vec{C})$, proving Part 2(ii) for the whole expression.

Since $e_3(x, y, \vec{X})$ has constant time complexity, the induction hypothesis of Part 1(iii) and (iv) is applicable. This gives us $gaifman(C'_j) \subseteq gaifman(C'_{j-1}, C_j, \vec{C})$, and for each $U \in molecule^0(C'_j)$, there are $V_{j,0}$, $V_{j,1}$, ..., V_{j,m_j} such that $atom^1(V_0) \subseteq atom^0(C'_{j-1}, C_j, \vec{C})$, $V_{j,i} \in molecule^0(C'_{j-1}, C_j, \vec{C})$ for $1 \leq i \leq m_j$, and $U = V_{j,0} \cup V_{j,1} \cup \cdots \cup V_{j,m_j}$. Note that $molecule^0(C'_{j-1}, C_j, \vec{C}) = molecule^0(C'_{j-1}) \cup molecule^0(\vec{C}) = molecule^0(C'_0) \cup molecule^0(\vec{C})$. Now, suppose $U \in molecule^0(C'_n)$, and $U = V_{n,0}$, $V_{n,1}$, ..., V_{n,m_n} such that $atom^1(V_0) \subseteq atom^0(C'_{n-1}, C_n, \vec{C}) = atom^0(C'_{n-1}) \cup atom^0(C_n) \cup atom^0(\vec{C}) = atom^0(C'_0) \cup atom^1(C'') \cup atom^0(\vec{C}) = A_0 \cup A'' \cup atom^0(\vec{C})$, $V_{n,i} \in molecule^0(C'_{n-1}, C_n, \vec{C}) = molecule^0(C'_0) \cup molecule^0(\vec{C})$, for $1 \leq i \leq m_n$.

Next, suppose $(u, v) \in gaifman(C'_n) = gaifman^0(C'_n) \cup gaifman^1(C'_n)$. Since $atom^0(C'_n) = A \cup atom^0(\vec{C})$, we have $(u, v) \in gaifman^0(C'_n)$ implies either $(u, v) \in gaifman^0(\vec{C})$, or $u \in A$ and $v \in atom^0(\vec{C})$, or $v \in A$ and $u \in atom^0(\vec{C})$, or u, $v \in A$. Since $A = A' \cup A'' \cup A_0 \subseteq atom^{\leq 1}(\vec{C})$, we get $(u, v) \in gaifman^0(C'_n)$ implies either $(u, v) \in gaifman^0(\vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $v \in atom^0(\vec{C})$ and $u \in atom^1(\vec{C})$, or $u \in A$ and $v \in atom^1(\vec{C})$, or $v \in A$ and $u \in atom^1(\vec{C})$. This proves Part 2(iii) when $(u, v) \in gaifman^0(C'_n)$. On the other hand, $(u, v) \in gaifman^1(C'_n)$ implies $(u, v) \in gaifman^1(C'_{n-1}, C_{n-1}, \vec{C}) = gaifman^1(C'_{n-1}) \cup gaifman^1(C_{n-1}) \cup gaifman^1(\vec{C}) \subseteq gaifman^1(C'_0) \cup gaifman^1(C_1) \cup \cdots \cup gaifman^1(C_{n-1}) \cup gaifman^1(\vec{C}) = gaifman^1(C'_0) \cup gaifman^1(\vec{C})$. Now Part 2(iii) for $(u, v) \in gaifman^1(C'_n)$ follows immediately by the induction hypothesis of Part 2(iii) on $e_1(\vec{X})$. This completes the proof of Part 2 for the second subcase when the *fold* expression has linear time complexity.

***

The next case is when $e(\vec{X})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in fold(x_2 := e_2(\vec{X}), y \in e_3(\vec{X}), e_4(x, y, \vec{X}))\}$. Suppose $fold(x_2 := e_2[\vec{C}/\vec{X}], y \in e_3[\vec{C}/vecX], e_4[\vec{C}/\vec{X}](x, y) \Downarrow C''$, and $\bigcup\{e_1[\vec{C}/\vec{X}](x)\ x \in Y[C''/Y]\} \Downarrow C'$. For Part 1, suppose the expression $\bigcup\{e_1(x, \vec{X})\ x \in Y\}$ has constant time complexity. By the induction hypothesis of Part 1 on the *fold* subexpression, we get $atom^0(C'') \subseteq atom^0(\vec{C})$, $atom^1(C'') \subseteq atom^{\leq 1}(\vec{C})$, $gaifman(C'') \subseteq gaifman(\vec{C})$, and for each $U \in molecule^0(C'')$, there are V_0, V_1, ..., V_m such that $atom^1(V_0) \subseteq atom^0(\vec{C})$, $V_j \in molecule^0(\vec{C})$ for $1 \leq j \leq m$, and $U = V_0 \cup V_1 \cup \cdots \cup V_m$. Then, by the induction hypothesis of Part 1(i) on $\{e_1(x, \vec{X}) \mid x \in Y\}$, we get $atom^0(C') \subseteq atom^0(C'', \vec{C}) = atom^0(C'') \cup atom^0(\vec{C}) = atom^0(\vec{C})$. By the induction hypothesis of Part 1(ii), we get $atom^1(C') \subseteq atom^{\leq 1}(C'', \vec{C}) = atom^{\leq 1}(C'') \cup atom^{\leq 1}(\vec{C}) = atom^{\leq 1}(\vec{C})$. By the induction hypothesis of Part 1(iii), we get $gaifman(C') \subseteq gaifman(C'', \vec{C}) \subseteq gaifman(\vec{C})$. Finally, by induction hypothesis of Part 1(iv), we get for each $U \in molecule^0(C')$, there are V_0, V_1, ..., V_m such that $atom^1(V_0) \subseteq atom^0(C'', \vec{C}) = atom^0(\vec{C})$, $V_j \in molecule^0(C'', \vec{C}) = molecule^0(C'') \cup molecule^0(\vec{C})$ for $1 \leq j \leq m$, and $U = V_0 \cup V_1 \cup \cdots \cup V_m$. This proves Part 1 for the whole expression.

For Part 2, suppose the expression $\bigcup\{e_1(x, \vec{X})\ x \in Y\}$ has linear time complexity. Note that if the *fold* subexpression has constant time complexity, so that properties Part 1(i)-(iv) hold for it, then properties Part 2(i)-(iii) hold for it as well. Thus, suffices to apply the induction hypothesis of Part 2 on the *fold* subexpression. This gives a number k'' and a set $A'' \subseteq atom \leq 1(\vec{C})$ where $|A''| \leq k''$;

and $atom^0(C'') \subseteq atom^0(\vec{C})$, $atom^1(C'') \subseteq atom^{\leq 1}(\vec{C})$, for each $(u, v) \in gaifman(C'')$, either $(u, v) \in gaifman(\vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $v \in atom^0(\vec{C})$ and $u \in atom^1(\vec{C})$, or $u \in A''$ and $v \in atom^1(\vec{C})$, or $v \in A''$ and $u \in atom^1(\vec{C})$. Then, by the induction hypothesis of Part 2(i) on $\{e_1(x, \vec{X}) \mid x \in Y\}$, we get $atom^0(C') \subseteq atom^0(C'', \vec{C}) = atom^0(C'') \cup atom^0(\vec{C}) = atom^0(\vec{C})$. By the induction hypothesis of Part 2(ii), we get $atom^1(C') \subseteq atom^{\leq 1}(C'', \vec{C}) = atom^{\leq 1}(C'') \cup atom^{\leq 1}(\vec{C}) = atom^{\leq 1}(\vec{C})$. And by the induction hypothesis of Part 2(iii), we get for each $(u, v) \in gaifman(C')$, either $(u, v) \in gaifman(C'', \vec{C}) = gaifman(C'') \cup gaifman(\vec{C})$, or $u \in atom^0(C'', \vec{C})$ and $v \in atom^1(C'', \vec{C})$, or $v \in atom^0(C'', \vec{C})$ and $u \in atom^1(C'', \vec{C})$, or $u \in A'$ and $v \in atom^1(C'', \vec{C})$, or $v \in A'$ and $u \in atom^1(C'', \vec{C})$, where $A' \subseteq atom^{\leq 1}(C'', \vec{C})$, $|A'| \leq k'$ as per the induction hypothesis.

Note that $atom^1(C'', \vec{C}) = atom^1(C'') \cup atom^1(\vec{C})$; $atom^1(C'') \subseteq atom^{\leq 1}(\vec{C}) = atom^0(\vec{C}) \cup atom^1(\vec{C})$; and u, $v \in atom^0(\vec{C})$ implies $(u, v) \in gaifman(\vec{C})$. Substituting these into the above, we get for each $(u, v) \in gaifman(C')$, either $(u, v) \in gaifman(C'') \cup gaifman(\vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $v \in atom^0(\vec{C})$ and $u \in atom^1(\vec{C})$, or $u \in A'$ and $v \in atom^1(\vec{C})$, or $v \in A'$ and $u \in atom^1(\vec{C})$. Now, setting $k = k' + k''$ and $A = A' \cup A''$, and using the fact that for each $(u, v) \in gaifman(C'')$, either $(u, v) \in gaifman(\vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $v \in atom^0(\vec{C})$ and $u \in atom^1(\vec{C})$, or $u \in A''$ and $v \in atom^1(\vec{C})$, or $v \in A''$ and $u \in atom^1(\vec{C})$, we conclude that for each $(u, v) \in gaifman(C')$, either $(u, v) \in gaifman(\vec{C})$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $v \in atom^0(\vec{C})$ and $u \in atom^1(\vec{C})$, or $u \in A$ and $v \in atom^1(\vec{C})$, or $v \in A$ and $u \in atom^1(\vec{C})$. This completes the proof of Part 2 for the case when $e(\vec{X})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in fold(x_2 := e_2, y \in e_3, e_4)\}$.

The remaining cases are similar to the proof of Lemma 3.1 and are omitted. □

Since Part 1 of Lemma 3.1 and 5.4 are identical, the proof that $head(x, X)$ requires at least linear time complexity in $\mathcal{NRC}_1(\leq)$ can be copied verbatim to $\mathcal{NRC}_1(\leq, fold, sort)$. This means, even when instantaneous sorting is available, $\mathcal{NRC}_1(\leq, fold, sort)$ cannot produce the head of a list (i.e. the first element of a canonical set) in constant time.

Corollary 5.5. *Let $head(x, X) : \{b_1 \times b_2\}$ be an expression in $\mathcal{NRC}_1(\leq, fold, sort)$. Suppose for every object c of type b_1, and non-empty canonical object C of type $\{b_1 \times b_2\}$ whose first element is (c, c_0), and c does not appear in subsequent elements of C. $e[c/x, C/X] \Downarrow \{(c, c_0)\}$. Then $time(head[C/X] \Downarrow)$ is at least $|C|$.*

Moreover, $\mathcal{NRC}_1(\leq, fold, sort)$ cannot realize a linear-time $zip(X, Y)$ for X and Y in canonical form, in spite of the availability of instantaneous sorting.

Corollary 5.6. *Let $zip(X, Y) : \{b_1 \times b_2\}$ be an expression in $\mathcal{NRC}_1(\leq, fold, sort)$ where X is a variable of type $\{b_3 \times b_1\}$, Y is a variable of type $\{b_3 \times b_2\}$, and b_1, b_2, and b_3 are distinct base types. Suppose for every canonical objects $U == \{(o_1, u_1), ..., (o_n, u_n)\}$ of type $\{b_3 \times b_1\}$ and $V == \{(o_1, v_1), ..., (o_n, v_n)\}$ of type $\{b_3 \times b_2\}$, $zip[U/X, V/Y] \Downarrow C'$ where $C' == \{(u_1, v_1), ..., (u_n, v_n)\}$. Then $time(zip[U/X, V/Y] \Downarrow)$ is at least $|U| * |V|$.*

Proof. Suppose $zip(X, Y)$ has linear time complexity. Then $gaifman(C') = C' = \{(u_1, v_1), ..., (u_n, v_n)\}$. However, for $1 \leq i \leq n$, $(u_i, v_i) \in gaifman(C') \notin gaifman(U, V) = U \cup V$. Then, by Part 2 of Lemma 5.4, either $u_i \in atom^0(U, V)$ or $v_i \in atom^0(U, V)$ or $u_i \in A$ or $v_i \in A$, for some $A \subseteq atom^1(U, V)$ and $|A|$ is independent of U and V. However, as U and V are both elists, $atom^0(U, V) = \{\}$ and thus contains neither u_i nor v_i. This means A has to contain every u_i and v_i. So, $|A|$ cannot be independent of U and V. This contradiction implies $zip(X, Y)$ cannot have linear time complexity.

The argument that $zip(X, Y)$ cannot have constant time complexity is same as the proof of Proposition 4.2. As $zip(X, Y)$ cannot have constant or linear time complexity, by Proposition 2.5, it has at best quadratic time complexity in $\mathcal{NRC}_1(\leq, \mathit{fold}, \mathit{sort})$. □

The limited-mixing lemma on $\mathcal{NRC}_1(\leq, \mathit{fold}, \mathit{sort})$, viz. Lemma 5.4, is worth a further remark. A reader familiar with Gaifman's locality property on first-order query languages [8] should notice the use of a variant of Gaifman graphs in limited-mixing lemmas here. Indeed, these limited-mixing lemmas can be regarded as intensional counterparts to Gaifman's locality property. Gaifman's locality property is often useful for analyzing the extensional expressive power [11, 13] and intensional express power [25] of query languages on unordered data types. However, it is quite useless on ordered data types, as the standard Gaifman graph becomes a complete graph on ordered data types causing all queries to have a locality index of 1. Also, Gaifman's locality property is unlikely to be useful in the presence of a *fold*-like function, as the standard Gaifman graph of the output of a query involving a *fold*-like function can become an arbitrarily long chain, causing the query's locality index to become infinite. In contrast, limited-mixing lemmas do not have either of these limitations. For example, the limited-mixing lemma on $\mathcal{NRC}_1(\leq, \mathit{fold}, \mathit{sort})$ remains easy to use as an off-the-self strategy for inexpressibility proofs; cf. Corollary 5.6.

5.3 Adding *zip*

In the preceding sections, *zip* was used an example of a low-selectivity join that cannot be realized efficiently in $\mathcal{NRC}_1(\leq)$, $\mathcal{NRC}_1(\leq, \mathit{takewhile}, \mathit{dropwhile}, \mathit{sort})$, and $\mathcal{NRC}_1(\leq, \mathit{fold}, \mathit{sort})$. In the most common implementation of $zip(X, Y)$, it has linear time complexity $\Theta(\min(|X|, |Y|))$, which we also assume here. Is adding *zip* to the query language sufficient to realise efficient low-selectivity joins? Unfortunately, a limited-mixing lemma also holds on $\mathcal{NRC}_1(\leq, \mathit{zip}, \mathit{sort})$.

Lemma 5.7 (Limited mixing). *Let $e(\vec{X}) : s$ be an expression in $\mathcal{NRC}_1(\leq, zip, sort)$. Suppose objects $\vec{C}$ have the same types as $\vec{X}$, and $e(\vec{C}/\vec{X}) \Downarrow C'$.*

(1) *If $e(\vec{X})$ has constant time complexity, then there is some number h that depends only on $e(\vec{X})$ but not on $\vec{C}$, and an undirected graph H where the nodes are a subset of $atom^{\leq 1}(\vec{C})$ and there are at most h edges in H, such that*
(i) $atom^0(C') \subseteq atom^0(\vec{C})$,
(ii) $atom^1(C') \subseteq atom^{\leq 1}(\vec{C})$,
(iii) $gaifman(C') \subseteq gaifman(\vec{C}) \cup H$, and
(iv) for each $U \in molecule^0(C')$, there are $V_0, V_1, ..., V_m$ such that $atom^1(V_0) \subseteq atom^0(\vec{C})$, $V_j \in molecule^0(\vec{C})$ or $gaifman(V_j) \subseteq H$ for each $1 \leq j \leq m$, and $U = V_0 \cup V_1 \cup \cdots \cup V_m$.

(2) *If $e(\vec{X})$ has linear time complexity, then there is a number k that depends only on $e(\vec{X})$ but not on $\vec{C}$, and an undirected graph K where the nodes are a subset of $atom^{\leq 1}(\vec{C})$ and each node w of K has degree at most nk, n is the number of times w appears in $\vec{C}$, such that*
(i) $atom^0(C') \subseteq atom^0(\vec{C})$,
(ii) $atom^1(C') \subseteq atom^{\leq 1}(\vec{C})$, and
(iii) for each $(u, v) \in gaifman(C')$, either $(u, v) \in gaifman(\vec{C}) \cup K$, or $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$, or $u \in atom^1(\vec{C})$ and $v \in atom^0(\vec{C})$.

Proof. Without loss of generality, $e(\vec{X})$ is assumed to be in the normal form induced by the restricted system of rewrite rules. The proof proceeds by induction on the structural complexity of expressions; i.e. the ordering $\sqsubseteq$ on expressions.

For Part 1, only the case when $e(\vec{X})$ has the form $zip(e_1, e_2)$ is elaborated below, as other forms of $e(\vec{X})$ have straightforward argument. Now, assume $zip(e_1, e_2)$ has contant time complexity. Then, both $e_1(\vec{X})$ and $e_2(\vec{X})$ has contant time complexity. Let $e_1(\vec{C}/\vec{X}) \Downarrow C'_1$. Let h_1 and H_1 be the number h and the graph H induced by Part 1 on $e_1(\vec{X})$. Let $e_2(\vec{C}/\vec{X}) \Downarrow C'_2$. Let h_2 and H_2 be the number h and the graph H induced by Part 1 on $e_2(\vec{X})$. By Part 1(i), $atom^0(C'_1)$, $atom^0(C'_2) \subseteq atom^0(\vec{C})$. Also, $atom^0(zip(C'_1, C'_2)) = \{\} \subseteq atom^0(\vec{C})$. By Part 1(ii), $atom^1(C'_1)$, $atom^1(C'_2) \subseteq atom^{\leq 1}(\vec{C})$. Then $atom^1(zip(C'_1, C'_2)) \subseteq atom^1(C'_1) \cup atom^1(C'_2) \subseteq atom^{\leq 1}(\vec{C})$. By Part 1(iii), $gaifman(C'_1) \subseteq gaifman(\vec{C}) \cup H_1$ and $gaifman(C'_2) \subseteq gaifman(\vec{C}) \cup H_2$. Let $C'_1 == V_{1,0} \cup V_{1,1} \cup \cdots \cup V_{1,m_1}$ and $C'_2 == V_{2,0} \cup V_{2,1} \cup \cdots \cup V_{2,m_1}$, as per Part 1(iv). Since $zip(e_1, e_2)$ has constant time complexity, it is safe to assume that $V_{i,j} \notin molecule^0(\vec{C})$ for $i \in \{1, 2\}$ and $1 \leq j \leq m_1$. Thus, $atom^1(V_{1,0}) \subseteq atom^0(\vec{C})$ has an upperbound on its cardinality that can be inferred from the types of $\vec{C}$; and for $1 \leq j \leq m_1$, $gaifman(V_{1,j}) \subseteq H_1$ implies an upperbound on $V_{1,j}$'s cardinality that can be inferred from h_1 and the type of C'_1. The sum of these upperbounds gives an upperbound on the cardinality of C'_1. An upperbound can be derived for the cardinality of C'_2 in a similar way. The mininum of these two upperbounds is an upperbound a on the cardinality of $zip(C'_1, C'_2)$. If the type of C'_1 is b^{k_1} and that of C'_2 is b^{k_2}, h can be set as $a(k_1 + k_2)!$ and H can be set to $gaifman(zip(C'_1, C'_2))$. Then Part 1(iii) and (iv) hold for $zip(C'_1, C'_2)$. This proves Part 1.

For Part 2, first consider the case when $e(\vec{X})$ has the form $zip(e_1(\vec{X}), e_2(\vec{X}))$ and has linear time complexity. Let $zip(e_1[\vec{C}/\vec{X}], e_2[\vec{C}/\vec{X}]) \Downarrow C'$. Then $atom^0(C') = \{\} \subseteq atom^0(\vec{C})$, proving Part 2(i) for the case. Let $e_1(\vec{C}/\vec{X}) \Downarrow C'_1$ and $e_2(\vec{C}/\vec{X}) \Downarrow C'_2$. By the induction hypothesis of Part 2 on e_1 and e_2, we get $atom^1(C'_1)$, $atom^1(C'_2) \subseteq atom^{\leq 1}(\vec{C})$. Thus, $atom^1(C') \subseteq atom^1(C'_1) \cup atom^1(C'_2) \subseteq atom^{\leq 1}(\vec{C})$, proving Part 2(ii) for the case. Now, suppose $(u, v) \in gaifman(C')$. There are five subcases. The first subcase is when $u \in atom^0(\vec{C})$ and $v \in atom^0(\vec{C})$, and thus $(u, v) \in gaifman(\vec{C})$. The second subcase is when $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$. The third subcase is when $u \in atom^1(\vec{C})$ and $v \in atom^0(\vec{C})$. The fourth subcase is when $u, v \in atom^1(\vec{C})$ and $(u, v) \in gaifman(\vec{C})$. These four subcases satisfy the requirement of Part 2(iii) trivially. The fifth and last subcase is when $u, v \in atom^1(\vec{C})$ and $(u, v) \notin gaifman(\vec{C})$. Let k_1 and K_1 be the k and K induced by the induction hypothesis of Part 2 on e_1. Let k_2 and K_2 be the k and K induced by the induction hypothesis of Part 2 on e_2. Suppose u appears n_u times and v appears n_v times in $\vec{C}$. Then, the induction hypothesis of Part 2 on $e_1(\vec{X})$ and $e_2(\vec{C})$ implies u appears at most $n_u k_1$ times in C'_1 and at most $n_u k_2$ times in C'_2, and v appears at most $n_v k_1$ times in C'_1 and at most $n_v k_2$ times in C'_2. Then, by definition of zip, the number of times that u can appear in C' is upperbounded by $n_u(k_1 + k_2)$, and that of v is upperbounded by $n_v(k_1 + k_2)$. Assume C'_1 has type b^{h_1} and C'_2 has type b_{h_2}. Thus, C' has type $b^{h_1+h_2}$. By the definition of zip, each occurrence of u in C' can add no more than $h_1 + h_2$ edges to $gaifman(C')$. Similarly, each occurrence of v in C' can add no more than $h_1 + h_2$ edges to $gaifman(C')$. Hence, setting $k = (k_1 + k_2)(h_1 + h_2)$ and $K = gaifman(C') - gaifman(\vec{C})$ is sufficient to ensure Part 2(iii). This completes the proof of Part 2 when $e(\vec{X})$ has the form $zip(e_1(\vec{X}), e_2(\vec{X}))$.

For Part 2, the next case to consider is when $e(\vec{X})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in zip(e_2(\vec{X}), e_3(\vec{X})\}$ and has linear time complexity. Suppose $e[\vec{C}/\vec{X}] \Downarrow C'$; $zip(e_2[\vec{C}/\vec{X}], e_3[\vec{C}/\vec{X}]) \Downarrow C''$, where $C'' == \{C''_1, ..., C''_m\}$; and $e_1[C''_i/x, \vec{C}/\vec{X}] \Downarrow C'_i$. There are two subcases. This first subcase is when m depends on $zip(e_2(\vec{X}), e_3(\vec{X}))$ but not on $\vec{C}$. Note that C'_i has set type; thus, $atom^0(C'_i) = \{\}$. By construction, $C' == C'_1 \cup \cdots \cup C'_m$. So, $atom^0(C') = \{\} \subseteq atom^0(\vec{C})$, proving Part 2(i). Also, C''_i is a tuple of base type. Then, by induction hypothesis of Part 2(ii) on the zip subexpression, we get $atom^{\leq 1}(C''_i) = atom^1(C''_i) \subseteq atom^{\leq 1}(\vec{C})$. By the induction hypothesis of Part 2(ii) on $e_1(x, \vec{X})$, we get $atom^1(C'_i) \subseteq atom^{\leq 1}(C''_i, \vec{C}) = atom^{\leq 1}(C''_i) \cup atom^{\leq 1}(\vec{C}) = atom^{\leq 1}(\vec{C})$, proving Part 2(ii). Let k' be the k induced by the induction hypothesis of Part 2 on $e_1(x, \vec{X})$. Let K_i be the K induced by the induction hypothesis of Part 2 on $e_1(x, \vec{X})$ when $e_1[C_i''/x, \vec{C}/\vec{X}] \Downarrow C_i'$. Now, suppose $(u, v) \in gaifman(C_i') \subseteq gaifman(C')$. By the induction hypothesis of Part 2, there are five possibilities. The first possibility is $u, v \in atom^0(\vec{C})$; thus $(u, v) \in gaifman(\vec{C})$. The second possibility is $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$. The third possibility is $u \in atom^1(\vec{C})$ and $v \in atom^0(\vec{C})$. The fourth possibility is u, $v \in atom^1(\vec{C})$ and $(u, v) \in gaifman(\vec{C})$. These four possibilities trivially meet the requirements for Part 2(iii). So, we focus on the fifth and last possibility, which is u, $v \in atom^1(\vec{C})$ and $(u, v) \notin gaifman(\vec{C})$. This implies $(u, v) \in K_i$. So, u has degree at most $n_u k'$ and v has degree at most $n_v k'$ in K_i, where n_u and n_v are the number of occurrences of u and v in $\vec{C}$. To prove Part 2(iii), it is sufficient to set $K = K_1 \cup \cdots \cup K_m$ and $k = mk'$.

The second subcase is when m depends on $\vec{C}$. This implies $e_1(x, \vec{X})$ has constant time complexity. Let h' be the h induced by the induction hypothesis of Part 1 on $e_1(x, \vec{X})$. Let H_i be the H induced by the induction hypothesis of Part 1 on $e_1(x, \vec{X})$ when $e_1[C_i''/x], \vec{C}/\vec{X}] \Downarrow C_i'$. By the induction hypothesis of Part 2, there are five possibilities for $(u, v) \in gaifman(C_i') \subseteq gaifman(C')$. The first possibility is u, $v \in atom^0(\vec{C})$; thus, $(u, v) \in gaifman(\vec{C})$. The second possibility is $u \in atom^0(\vec{C})$ and $v \in atom^1(\vec{C})$. The third possibility is $u \in atom^1(\vec{C})$ and $v \in atom^0(\vec{C})$. The fourth possibility is u, $v \in atom^1(\vec{C})$, and $(u, v) \in gaifman(\vec{C})$. These four possibilities already meet the requirements for Part 2(iii). So, we focus on the fiftth and last possibility, which is u, $v \in atom^1(\vec{C})$, and $(u, v) \notin gaifman(\vec{C})$. This implies $(u, v) \in H_i$. Let $K = H_1 \cup \cdots \cup H_m$. Let k' be the k induced by the induction hypothesis of Part 2 on the *zip* subexpression. Then, u can appear no more than $n_u k'$ times in C'', where n_u is the number of times u appears in $\vec{C}$. So, u has degree at most $n_u k' h'$ in K; similarly, v has degree at most $n_v k' h'$ in K. Thus, setting $k = k'h'$ and $K = H_1 \cup \cdots \cup H_m$ suffices to prove Part 2(iii) for this subcase. Part 2(i) and Part 2(ii) have similar argument as in the previous subcase. This completes the proof of Part 2 for this subcase.

The remaining cases for Part 2 are straightforward, except for the case when $e(\vec{X})$ has the form $\bigcup\{e_1(x, \vec{X}) \mid x \in e_2(\vec{X})\}$. However, the argument for this case can be copied almost verbatim from the subcase above. This finishes the proof of the lemma. □

Consider the following query, $contrived(X, Y) =_{df} \{(x, y_1, y_2) \mid x \in X, (y_1, y_2) \in Y, y_1 < x < y_2\}$, where X and Y are in canonical form. Suppose $Y == \{(y_{1,1}, y_{1,2}), (y_{2,1}, y_{2,2}), ..., (y_{n,1}, y_{n,2})\}$ is such that $y_{i,j}$ are all positive numbers, $y_{i,1} < y_{i+1,1}$, and no number is allowed to appear more than once in Y. Under these constraints, a number can appear at most once in X and once in Y. A further low-selectivity constraint—let us call this the k-selectivity constraint—is imposed such that no element in X is contained by more than k intervals in Y, and no interval in Y can contain more than k elements in X, for some fixed k. It is obvious that $contrived(X, Y)$, even when the k-selectivity contraint is imposed on (X, Y), cannot be implemented in linear time in $\mathcal{NRC}_1(\leq, zip, sort)$.

Proposition 5.8. *Let $f(X, Y)$ be an expression in $\mathcal{NRC}_1(\leq, zip, sort)$. Then there is a number k_0 such that either $f(X, Y)$ does not have linear time complexity or $f(X, Y) \neq contrived(X, Y)$ under the k-selectivity constraint for all $k > k_0$.*

Later, in Example 6.6, an efficient solution for a variant of *contrived* under such a low-selectivity constraint will be presented.

6 SYNCHRONY ITERATORS

6.1 Capturing the gap

Although neither *takewhile* and *dropwhile* nor *fold* is able to fill the intensional expressiveness gap between $\mathcal{NRC}_1(\leq)$ and relational database systems, using all three of them simultaneously can do the job. To see this, consider the construct below:

$$\frac{X : \{s_1\} \quad Y : \{s_2\} \quad bf : \mathbb{B} \quad cs : \mathbb{B} \quad f : \{s\}}{syncmap(x^{s_1} \in X, y^{s_2} \in ys^{\{s_2\}} \subseteq Y, bf, cs, f) : \{s\}}$$

where the notations $x \in X$ and $y \in ys \subseteq Y$ introduce fresh variables x, y, and ys; the scopes of these variables are as indicated in the following: $bf(y, x)$, $cs(y, x)$, and $f(x, ys)$. Note that X, Y, $bf(y, x)$, $cs(y, x)$, and $f(x, ys)$ are expressions; these are used here (instead of the more generic e) for clarity purpose. Also, for simplifying proofs later, x and y are the only free variables allowed for bf and cs. This *syncmap* construct is defined in terms of *takewhile*, *dropwhile*, and *fold* as $syncmap(x \in X, y \in ys \subseteq Y, bf, cs, f) =_{df} fold(a := (Y, \{\}), x \in X, (\{u \mid u \in takewhile(y \in a.\pi_1, bf(y, x)\ or\ cs(y, x)), cs(u, x)\} \cup dropwhile(y \in a.\pi_1, bf(y, x)\ or\ cs(y, x)), a.\pi_2 \cup f(x, \{u \mid u \in takewhile(y \in a.\pi_1, bf(y, x)\ or\ cs(y, x)), cs(u, x)\}))).\pi_2$. As *syncmap* relies on *takewhile*, *dropwhile*, and *fold*, to ensure soundness, its X and Y inputs are required to be in canonical form. Again, *sort* is available for explicitly putting the inputs to *syncmap* into canonical form when needed.

According to the definition above, $syncmap(x \in X, y \in ys \subseteq Y, bf, cs, f)$ can be seen as an iteration on Y in synchrony with an iteration on X. For each $x_i \in X$, it iterates on Y from the current $y_j \in Y$, as long as $bf(y_j, x_i)$ or $cs(y_j, x_i)$ holds. Let ys_i be an elist consisting of those y_j seen for this x_i such that $cs(y_j, x_i)$ holds. The result $C_i == f(x_i, ys_i)$ is computed. All the y_j seen for this x_i, except those in ys_i, are dropped from Y. The iteration on X is moved on to the next element in X, while the iteration on Y is resumed from the start of the modified Y. It can be seen from the *takewhile* and *dropwhile* subexpressions that for any x_i, the iteration on Y does not move beyond the first y_j in Y such that both $bf(y_j, x_i)$ and $cs(y_j, x_i)$ are *false*. And for any x_i, the iteration on Y starts directly at the first y_j in Y such that $cs(y_j, x_{j-1})$ is *true*. This means that, when the $y_j \in Y$ for which $cs(y_j, x_i)$ holds are clustered near each other in Y for every $x_i \in X$, the iteration on Y effectively jumps directly to the cluster of each x_i. In this sense, the iteration on X and Y can be said to be synchronized. At the end of this synchronized iteration, $\bigcup_i C_i$ is produced as the result. It is worth noting that $\cup$ is realized using $\oplus$, the concatenation operation on elists, which has constant time complexity.

It turns out that synchronized iteration is the missing ingredient from $\mathcal{NRC}_1(\leq)$ that results in the intensional expressiveness gap between $\mathcal{NRC}_1(\leq)$ and relational database systems. This is proved as Theorem 6.3 after some relevant definitions are given below.

Definition 6.1 (Monotonicity). For x_i and x_j in an elist $L == \{x_1, ..., x_n\}$, let $(x_i \ll x_j \mid L)$ means x_i is before x_j in L; i.e. $(x_i \ll x_j \mid L)$ iff $i < j$. A predicate $bf(y, x)$ is said to be monotonic with respect to elists X and Y if it satisfies these conditions: (i) If $(x \ll x' \mid X)$, then for each y in Y, $bf(y, x)$ implies $bf(y, x')$. (ii) If $(y' \ll y \mid Y)$, then for each x in X, $bf(y, x)$ implies $bf(y', x)$.

Definition 6.2 (Antimonotonicity). Suppose $bf(y, x)$ is a monotonic predicate with respect to elists X and Y. A predicate $cs(y, x)$ is said to be antimonotonic with respect to $bf(y, x)$ if it satisfies these conditions: (i) If $(x \ll x' \mid X)$, then for each y in Y, $bf(y, x)$ and not $cs(y, x)$ implies not $cs(y, x')$. (ii) If $(y \ll y' \mid Y)$, then for each x in X, not $bf(y, x)$ and not $cs(y, x)$ implies not $cs(y', x)$.

Theorem 6.3 (Intensional expressiveness). *Let $bf(y, x)$ be monotonic with respect to X and Y, X and Y are in canonical form, and $cs(y, x)$ antimonotonic with respect to $bf(y, x)$.*

(1) *$syncmap(x \in X, y \in ys \subseteq Y, bf, cs, f) =$
$\{u \mid x \in X, u \in f(x, \{y \mid y \in Y, cs(y, x)\})\}$.*
(2) *$time(syncmap(x \in X, y \in ys \subseteq Y, bf, cs, f)) = O(|X| + 3(k+1)|Y| + g(k|Y|/|X|)|X|)$, provided there is some k such that for each y in Y, $|\{x \mid x \in X, cs(y, x)\}| \leq k$; $|\{y \mid y \in Y, cs(y, x_i)\}| \approx |\{y \mid y \in Y, cs(y, x_j)\}|$ for any $x_i, x_j \in X$; $time(f(x, ys)) = O(g(|ys|))$; and $bf(y, x)$ and $cs(y, x)$ have constant time complexity.*

Proof. Let $syncmap(x \in X, y \in ys \subseteq Y, bf, cs, f) \Downarrow C$. Let $X == \{x_1, ..., x_n\}$. Let $C_0 == \{\}$ and $E_0 == Y$. For $1 \leq j \leq n$, let
$A_j == takewhile(y \in E_{j-1}, bf(y, x_j) \textit{ or } cs(y, x_j))$;
$B_j == \{u \mid u \in A_j, cs(y, x_j)\}$;
$C_j == f(x_j, B_j)$;
$D_j == dropwhile(y \in E_{j-1}, bf(y, x_j) \textit{ or } cs(y, x_j))$;
$E_j == B_j \oplus D_j$; and
$a_j == (E_j, C_j)$.
Then it is easy to see that $C == C_1 \oplus \cdots \oplus C_n$.

Part 1 of the theorem can be proved by induction on $j > 0$ that
(i) $\{y \mid y \in Y, cs(y, x_j)\} \subseteq E_{j-1}$;
(ii) $\{y \mid y \in Y, cs(y, x_j)\} \subseteq A_j$;
(iii) $B_j == \{y \mid y \in Y, cs(y, x_j)\}$;
(iv) $C_j == f(x_j, \{y \mid y \in Y, cs(y, x_j)\})$;
(v) $\{y \mid y \in D_j, cs(y, x_j)\} = \{\}$; and
(vi) $\{y \mid y \in E_k - E_j, cs(y, x_j)\} = \{\}$ for $k < j$.

By the definition of D_j, if it is not empty, its first element must be a y such that both $bf(y, x_j)$ and $cs(y, x_j)$ are false. By construction, $(y \ll y' \mid Y)$ for all subsequent elements y' in D_j. Then, by the antimonotonicity of $cs(y, x)$, both $bf(y, x_j)$ and $cs(y, x_j)$ are false for each y in D_j. This proves Part (v) of the induction above.

By Part (i), $\{y \mid y \in Y, cs(y, x_j)\} \subseteq E_{j-1}$; By construction, $E_{j-1} == A_j \oplus D_j$. So, $\{y \mid y \in Y, cs(y, x_j)\} \subseteq A_j$, proving Part (ii). Then $B_j == \{y \mid y \in Y, cs(y, x_j)\}$ follows from the definition of B_j, proving Part (iii). Then $C_j == f(x_j, \{y \mid y \in Y, cs(y, x_j)\})$ follows from the definition of C_j, proving Part (iv).

By definition, $E_j == B_j \oplus D_j$. But $B_j == \{y \mid y \in Y, cs(y, x_j)\}$, as shown above for Part (iii). Thus, $E_k - E_j$ cannot contain any element y such that $cs(y, x_j)$, proving Part (vi).

$\{y \mid y \in Y, cs(y, x_1)\} \subseteq E_0$ because $E_0 == Y$, proving the base case (i.e. $j = 1$) for Part (i). Now, by the induction hypothesis on Part (vi), there is no y in $E_k - E_{j-1}$, $k < j - 1$, such that $cs(y, x_{j-1})$. By construction, $(x_{j-1} \ll x_j \mid X)$. Thus, by the antimonotonocity of $cs(y, x)$, for every $k < j - 1$, there is no y in $E_k - E_{j-1}$ such that $cs(y, x_j)$. Then $\{y \mid y \in Y, cs(y, x_j)\} \subseteq E_{j-1}$ follows from the fact that $Y = E_{j-1} \cup \bigcup_{k<j-1} E_k$. This settles Part (i) of the induction, and thus Part 1 of the theorem.

For Part 2 of the theorem, the time complexity of $syncmap(x \in X, y \in ys \subseteq Y, bf, cs, f)$ is the sum of the time complexity of computing each A_j, B_j, C_j, D_j, E_j, and a_j, plus the time taken for accessing each $x_j \in X$. As $bf(y, x)$ and $cs(y, x)$ are assumed to be constant time, the time complexity for computing A_j, B_j, and D_j is $O(|A|)$. The time complexity for computing E_j is constant time as $\oplus$ is assumed to have constant time complexity. The time complexity for computing a_j is constant time as tuple construction is, given that the components have already been constructed. The time complexity for C_j depends on $|B_j|$, as f's time complexity is assumed to depend on this. By assumption of the theorem, for each y in Y, $|\{x \mid x \in X, cs(y, x)\}| \leq k$. This means each y can appear in at most k different B_j. So, the total length of the B_j's is at most $k|Y|$; and the average length of the B_j's is at most $k|Y|/|X|$, as $|\{y \mid y \in Y, cs(y, x_i)\}| \approx |\{y \mid y \in Y, cs(y, x_j)\}|$ for x_i, x_j in X. The total length of the A_j's is $|E_0| + \sum_j |B_j| = |Y| + k|Y| = (k + 1)|Y|$. So, including the time for accessing each $x_j \in X$, the total time complexity is $|X| + 3(k + 1)|Y| + g(k|Y|/|X|)|X|$. □

Part 1 of Theorem 6.3 states that, for $bf(y, x)$ and $cs(y, x)$ monotonic and antimonotonic, *syncmap*—as a function—is already expressible using $\mathcal{NRC}_1(\leq)$ in a simple way as $\{u \mid x \in X, u \in f(x, \{y \mid y \in Y, cs(y, x)\})\}$. This equivalent version in comprehension syntax has $\Omega(|X| * |Y|)$ time complexity. In contrast, Part 2 of the theorem says that the algorithm corresponding to *syncmap* has time complexity $O(|X| + 3(k+1)|Y| + g(k|Y|/|X|)|X|)$. If f has constant time complexity with respect to its first parameter and linear time complexity with respect to its second parameter, this simplifies to $O(|X| + k|Y|)$, where k is the selectivity of $cs(y, x)$ which corresponds to the join condition.

To better appreciate this, let $X \bowtie Y =_{df} syncmap(x \in X, y \in ys \subseteq Y, y.\pi_1 < x.\pi_1, y.\pi_1 = x.\pi_1, \{(x.\pi_2, y.\pi_2) \mid y \in ys\})$. By Part 1 of Theorem 6.3, $X \bowtie Y = \{u \mid x \in X, u \in \{(x.\pi_2, y.\pi_2) \mid y \in \{y \mid y \in Y, y.\pi_1 = x.\pi_1\}\}\} = \{(x.\pi_2, y.\pi_2) \mid x \in X, y \in Y, y.\pi_1 = x.\pi_1\}$, which is of course a relational join of X and Y. By Part 2 of Theorem 6.3, $time(X \bowtie Y) = O(|X| + k|Y|)$, when $|\{x \mid x \in X, y.\pi_1 = x.\pi_1\}| < k$ for each $y \in Y$; i.e. k is the selectivity of the join. Therefore, for low-selectivity joins, i.e. k is a small number, the algorithm corresponding to *syncmap* has linear time complexity $O(|X| + |Y|)$. For high-selectivity joins, i.e. $k \approx |X|$, the algorithm has the usual quadratic time complexity $O(|X| * |Y|)$ of high-selectivity joins.

It was pointed out earlier that an intensional gap might also exist on relational intersection and relational difference. This becomes a non-issue as both relational intersection and relational difference can be realised with linear time complexity using *syncmap*. To wit, for canonical X and Y, let $X \cap Y =_{df} syncmap(x \in X, y \in ys \subseteq Y, y < x, y = x, \{x \mid not\ ys\ isempty\})$. By Part 1 of Theorem 6.3, $X \cap Y = \{u \mid x \in X, u \in \{x \mid not\ \{y \mid y \in Y, y = x\}\ isempty\}\} = \{x \mid x \in X, not\ \{y \mid y \in Y, y = x\} isempty\}$, which is of course the relational intersection of X and Y. By Part 2 of Theorem 6.3, $time(X \cap Y) = O(|X| + |Y|)$, as $f(x, ys) = \{x \mid not\ ys\ isempty\}$ has constant time complexity; i.e., a linear-time complexity relational intersection is realized using *syncmap*. Similarly, for canonical X and Y, a linear-time complexity relational difference can be realized as $X - Y =_{df} syncmap(x \in X, y \in ys \subseteq Y, y < x, y = x, \{x \mid ys\ isempty\})$.

It was also pointed out earlier that relational division $X \div Y$ is not directly supported by typical relational database systems. Instead, it is defined in terms of other operators supported by these systems. Leinders and Van den Bussche [12] showed that, if $X \div Y$ is expressed using strictly other operators from the relational algebra, then the space complexity is $\Omega(|X| * |Y|)$. This quadratic-space lower bound can be avoided using *syncmap*, though the time complexity remains quadratic. To see this, consider defining $X \div Y =_{df} syncmap(xb \in \{x.\pi_1 \mid x \in X\}, x \in xs \subseteq X, x.\pi_1 < xb, x.\pi_1 = xb, \{xb \mid Y \subseteq \{x.\pi_2 \mid x \in xs\}\})$. If $X : b \times a$ and $Y : a$ are canonical, the time complexity of $X \div Y$ as defined above is $O(|A| + |X| + |A| * (average(|xs|) + |Y|))$ where $A =_{df} \{x.\pi_1 \mid x \in X\}$, since $time(Y \subseteq \{x.\pi_2 \mid x \in xs\}) = O(|Y| + |xs|)$. When $|A|$ is at most circa $|X|/|Y|$, the average size of $|xs|$ is at most circa $|Y|$; in this case, the time complexity of $X \div Y$ becomes linear. However, $|A|$ can be $|X|$ in the worst case; then the time complexity of $X \div Y$ is quadratic, as in typical relational database systems. Space complexity has not been defined explicitly in the operational semantics of $\mathcal{NRC}$ so far. However, since space can be reused, the space complexity of evaluating $e(\vec{C}) \Downarrow$ can be taken as the space needed by the largest node in the evaluation tree $e(\vec{C}) \Downarrow$. Generally, the space needed by a node $e(\vec{C}) \Downarrow C'$ is $size(C')$ plus the sum of the space needed by the branches at this node. An exception to this definition of space needed is when the node is $fold(x := e_1, y \in e_2, e_3) \Downarrow C'_n$. For this node, according to Figure 4, the branches are $e_1 \Downarrow C'_0$, $e_2 \Downarrow \{C_1, ..., C_n\}$, $e_3[C'_0/x, C_1/y] \Downarrow C'_1$, ..., $e_3[C'_{n-1}/x, C_n/y] \Downarrow C'_n$. The space of C'_{j-1} can be reused once C_j is computed. So the space needed for evaluating the node $fold(x := e_1, y \in e_2, e_3) \Downarrow C'_n$ is $size(C'_0) + \sum_j size(C_j) + \max_j size(C'_j)$. Under this definition of space usage, it can be shown that the space required for evaluating $X \div Y$ is linear. Thus, while $X \div Y$ as defined above cannot escape the quadratic time-complexity lower bound, it can do better than the quadratic space-complexity lower bound required when relational division is compelled to be defined strictly using other relational operators [12].

So, $syncmap(x \in X, y \in ys \subseteq Y, bf, cs, f)$, constrained to $bf(y, x)$ being monotonic and $cs(y, x)$ being antimonotonic with respect to X and Y, bridges the intensional expressiveness gap of comprehension syntax while being a conservative extension to $\mathcal{NRC}_1(\leq)$. In short, *syncmap* under the monotonicity and antimonotonicity contraints is an exact characterization of this gap.

It is straightforward to generalize *syncmap* to synchronize iterations on h different elists to an iteration on a common reference elist. For example, to synchronize the iterations on two elists Y_1 and Y_2 to an iteration on an elist X, define the $syncmap_2$ construct as follows: $syncmap_2(x \in X, y_1 \in ys_1 \subseteq Y_1, bf_1, cs_1, y_2 \in ys_2 \subseteq Y_2, bf_2, cs_2, f) =_{df} fold(a := (Y_1, Y_2, \{\}), x \in X, (Y_1'(x, a.\pi_1), Y_2'(x, a.\pi_2), a.\pi_3 \cup f(x, ys_1(x, a.\pi_1), ys_2(x, a.\pi_2)))).\pi_3$, where $ys_i(x, Y) =_{df} \{u \mid u \in takewhile(y \in Y, bf_i(y, x)\ or\ cs_i(y, x)), cs_i(u, x)\}$, and $Y_i'(x, Y) =_{df} ys_i(x, Y) \cup dropwhile(y \in Y, bf_i(y, x)\ or\ cs_i(y, x))$. Note that $syncmap_2$, $syncmap_3$, etc. are actually definable solely in terms of *syncmap*, albeit tediously. So, for convenience, $\mathcal{NRC}_1(\leq, syncmap, sort)$ is defined here as $\mathcal{NRC}_1(\leq, sort)$ endowed with synchronized iteration constructs for multiple elists, viz. *syncmap*, $syncmap_2$, $syncmap_3$, etc. with the monotonicity and antimonotonicity constraints on their use.

Returning to the *head* query from Proposition 4.1; it is expressible as $head(x, X) =_{df} syncmap(u \in \{x\}, v \in vs \subseteq X, v.\pi_1 < u, v.\pi_1 = u, vs)$. To see that this definition of *head* has constant time complexity, substitute *syncmap* by its definition, this becomes $head(x, X) =_{df} fold(a := (X, \{\}), u \in \{x\}, (\{v \mid v \in takewhile(y \in a.\pi_1, y.\pi_1 < u\ or\ y.\pi_1 = u), v.\pi_1 = u\} \cup dropwhile(v \in a.\pi_1, v.\pi_1 < u\ or\ v.\pi_1 = u), a.\pi_2 \cup \{v \mid v \in takewhile(y \in a.\pi_1, y.\pi_1 < u\ or\ y.\pi_1 = u), v.\pi_1 = u\})).\pi_2$. Substitute *fold* by its definition, this becomes $head(x, X) =_{df} \{v \mid v \in takewhile(y \in X, y.\pi_1 < x\ or\ y.\pi_1 = x), v.\pi_1 = x\}$. Then, by the requirement of *head* in Proposition 4.1, and the operational semantics of *takewhile*, this definition of *head* has constant time complexity, as desired.

Similarly, the *zip* query from Proposition 4.2 is expressible as $zip(X, Y) =_{df} syncmap(u \in X, v \in vs \subseteq Y, v.\pi_1 < u.\pi_1, v.\pi_1 = u.\pi_1, \{(u, w) \mid w \in vs\})$. By Part 2 of Theorem 6.3, the time complexity of this definition of *zip* is $O(|X| + |Y|)$, i.e., linear, as desired.

6.2 Dovetailing into comprehension syntax

As shown earlier, *syncmap* characterizes the intensional expressiveness gap between $\mathcal{NRC}_1(\leq)$ and typical relational database systems. However, efficient joins in $\mathcal{NRC}_1(\leq, syncmap, sort)$ are no longer in comprehension syntax. This problem is addressed in this subsection.

Suppose there is an environment Γ, which is an updatable mapping from pointers to elists; i.e. Γ is a global store for elists. Let ι be a pointer into the global store. Let $\Gamma(\iota)$ means retrieving the elist C that ι points to in Γ. Let $\Gamma \oplus [\iota \mapsto C]$ denote the updated global store Γ' such that $\Gamma'(\iota) = C$ and $\Gamma'(\iota') = \Gamma(\iota')$ for $\iota \neq \iota'$. Let $\Gamma \ominus \iota$ denote the updated global store Γ' such that $\Gamma'(\iota)$ is undefined and $\Gamma'(\iota') = \Gamma(\iota')$ for $\iota \neq \iota'$.

A pair of constructs—*eiterator* and *syncedwith*—are given in Figure 5 to make use of the global updatable store. The construct *let* ι := *eiterator* e_1 *in* e_2 introduces a pointer ι whose scope is in the expression $e_2(\iota)$. Operationally, beginning with a global store Γ, the expression e_1 is evaluated to an object C_1 while the global store is updated to $\Gamma_1 = \Gamma \oplus [\iota \mapsto C_1]$. Then e_2 is evaluated with Γ_1, to produce an object C_2 while the global store is updated to Γ_2. Finally, C_2 is returned as the result, and the global store is updated to $\Gamma_2 \ominus \iota$ by removing ι.

The construct *let* ys := *syncedwith*$(x := e_1, y \in \iota, bf, cs)$ *in* e_2 introduces the variables x and y whose scope is the expressions $bf(y, x)$ and $cs(y, x)$, and the variable ys whose scope is the expression $e_2(ys)$. Operationally, beginning with a global store Γ, the elist $\Gamma(\iota) == \{u_1, ..., u_m\}$ is retrieved. The expression e_1 is evaluated with Γ to an object C while the global store is updated to Γ_1. Then beginning with $i = 1$ and Γ_1, the expressions $bf(u_i, C)$ is evaluated in Γ_i to produce an object v_i while updating Γ_i to Γ_i'; and the expression $cs(u_i, C)$ is evaluated in Γ_i' to produce an object v_i' while updating Γ_i' to Γ_{i+1}. This process is repeated until the first i such that $v_i = false$ and $v_i' = false$, if such an i exists. At this point, let $C' == \{u_j \mid 1 \leq j < i, v_j' = true\}$. And ys takes on as

Expression constructs for Synchrony iterators

$$\frac{e_1 : \{s_1\} \quad e_2 : s_2}{\mathit{let}\ \iota^{s_1} := \mathit{eiterator}\ e_1\ \mathit{in}\ e_2 : s_2}$$

$$\frac{e_1 : s_1 \quad bf : \mathbb{B} \quad cs : \mathbb{B} \quad e_2 : s_2}{\mathit{let}\ ys^{\{s_1\}} := \mathit{syncedwith}(x^{s_1} := e_1,\ y^{s_2} \in \iota^{s_2},\ bf,\ cs)\ \mathit{in}\ e_2 : s_2}$$

Operational semantics for Synchrony iterators

$$\frac{\Gamma, e_1 \Downarrow \Gamma_1, C_1 \quad \Gamma_1 \oplus [\iota \mapsto C_1], e_2 \Downarrow \Gamma_2, C_2}{\Gamma, \mathit{let}\ \iota := \mathit{eiterator}\ e_1\ \mathit{in}\ e_2 \Downarrow \Gamma_2 \ominus \iota, C_2}$$

$$\frac{\begin{array}{c}\Gamma, e_1 \Downarrow \Gamma_1, C \\ \Gamma_1, bf[u_1/y, C/x] \Downarrow \Gamma_1', v_1 \quad \Gamma_1', cs[u_1/y, C/x] \Downarrow \Gamma_2, v_1' \quad \cdots \\ \Gamma_i, bf[u_i/y, C/x] \Downarrow \Gamma_i', v_i \quad \Gamma_i', cs[u_i/y, C/x] \Downarrow \Gamma_{i+1}, v_i' \\ \Gamma_{i+1} \oplus [\iota \mapsto C' \oplus \{u_i, ..., u_m\}], e_2[C'/ys] \Downarrow \Gamma'', C''\end{array}}{\Gamma, \mathit{let}\ ys := \mathit{syncedwith}(x := e_1,\ y \in \iota,\ bf,\ cs)\ \mathit{in}\ e_2 \Downarrow \Gamma'', C''}$$

where $\Gamma(\iota) == \{u_1, ..., u_m\}$;
$v_j = \mathit{true}$ or $v_j' = \mathit{true}$ for $1 \leq j < i$;
$v_i = \mathit{false}$ and $v_i' = \mathit{false}$; and
$C' == \{u_j \mid 1 \leq j < i, v_j' = \mathit{true}\}$.

$$\frac{\begin{array}{c}\Gamma, e_1 \Downarrow \Gamma_1, C \\ \Gamma_1, bf[u_1/y, C/x] \Downarrow \Gamma_1', v_1 \quad \Gamma_1', cs[u_1/y, C/x] \Downarrow \Gamma_2, v_2' \quad \cdots \\ \Gamma_m, bf[u_m/y, C/x] \Downarrow \Gamma_m', v_m \quad \Gamma_m', cs[u_m/y, C/x] \Downarrow \Gamma_{m+1}, v_m' \\ \Gamma_{m+1} \oplus [\iota \mapsto C'], e_2[C'/ys] \Downarrow \Gamma'', C''\end{array}}{\Gamma, \mathit{let}\ ys := \mathit{syncedwith}(x := e_1,\ y \in \iota,\ bf,\ cs)\ \mathit{in}\ e_2 \Downarrow \Gamma'', C''}$$

where $\Gamma(\iota) == \{u_1, ..., u_m\}$;
$v_j = \mathit{true}$ or $v_j' = \mathit{true}$ for $1 \leq j \leq m$; and
$C' == \{u_j \mid 1 \leq j \leq m, v_j' = \mathit{true}\}$.

Fig. 5. The syntax and operational semantics of Synchrony iterators.

its value C', while the global store is updated to $\Gamma_i \oplus [\iota \mapsto C' \oplus \{u_i, ..., u_m\}]$. On the other hand, when there is no i such that $v_i = \mathit{false}$ and $v_i' = \mathit{false}$, the process is repeated until the last element u_m is processed. At this point, let $C' == \{u_j \mid 1 \leq j \leq m, v_j' = \mathit{true}\}$. And ys takes on as its value C', while the global store is updated to $\Gamma_{m+1} \oplus [\iota \mapsto C']$. Finally, $e_2(ys)$ is evaluated in this global store to produce the result C'', while updating the global store to Γ''.

For the purpose of this paper, viz. to simplify proofs and to reduce programming error, several "safe-use" constraints are imposed.

Definition 6.4 (Safe-use constraints). Expressions are required to satisfy the constraints below.

(1) The pointer ι introduced in *let* $\iota :=$ *eiterator* e_1 *in* e_2 is permitted to occur exactly once in the expression e_2.
(2) The variables x and y are the only free variables of bf and cs in *let* $ys :=$ *syncedwith*$(x := e_1, y \in \iota, bf, cs)$ *in* e_2.
(3) The expression e_1 in *let* $ys :=$ *syncedwith*$(x := e_1, y \in \iota, bf, cs)$ *in* e_2 is restricted to be a variable.
(4) The *eiterator* and *syncedwith* constructs must appear in the form: *let* $\iota_1 :=$ *eiterator* e_1 *in* ... *let* $\iota_h :=$ *eiterator* e_h *in* $\bigcup\{$ *let* $ys_1 :=$ *syncedwith*$(u_1 := x, y_1 \in \iota_1, bf_1, cs_1)$ *in* ... *let* $ys_h :=$ *syncedwith*$(u_h := x, y_h \in \iota_h, bf_h, cs_h)$ *in* $f(x, ys_1, ..., ys_h) \mid x \in e\}$.
(5) Furthermore, in the above, $bf_i(y, x)$ is monotonic with respect to e_i and e, $cs_i(y, x)$ is antimonotonic with respect to $bf_i(y, x)$, for $1 \leq i \leq h$.

The *eiterator* and *syncedwith* constructs have side effects that change the global store. These safe-use constraints are designed in part to isolate these side effects, so that analysis can be simplified. Constraint 1 and 4 together ensure that side effects due to $\iota_1, ..., \iota_h$ have no impact on the subexpression $f(x, ys_1, ..., ys_h)$ mentioned in Constraint 4. Notice that, by Constraint 1, $\iota_1, ..., \iota_h$ do not appear inside f; similarly, Constraint 2, implies none of $\iota_1, ..., \iota_h$ appear in bf and cs. Hence, even though $\iota_1, ..., \iota_h$ are in the global store for f, bf, and cs to access when they are evaluated, they actually do not access any of these pointers; so, they cannot make changes to or be by affected by changes made to what these pointers are pointing to. In other words, Constraint 1, 2, and 4 collectively make it possible for bf, cs, and f to be analyzed as if there is no side effect. Constraint 3 is just a syntactic restriction to make it easier to state Constraint 4. Finally, Constraint 5 reflects the key assumptions required for *syncmap* to work properly, cf. Theorem 6.3; as will be seen shortly in Theorem 6.5, these assumptions are also needed for *syncedwith* to work properly.

The syntactic sugar *let* $ys :=$ *syncedwith*$(x, y \in \iota, bf, cs)$ *in* e is used as a shortand for *let* $ys :=$ *syncedwith*$(u := x, y \in \iota, bf[u/x], cs[u/x])$ *in* e. And the syntactic sugar *syncedwith*$(x, y \in \iota, bf, cs)$ is used as a shorthand for *let* $ys :=$ *syncedwith*$(x, y \in \iota, bf, cs)$ *in* ys.

The translation for comprehension syntax $\{e \mid \delta_1, ..., \delta_n\}$ is also extended so that δ_i is permitted to take the additional forms $\iota :=$ *eiterator* e' and $ys :=$ *syncedwith*$(x := e_1, y \in \iota, bf, cs)$. The required extra translation rules are $\{e \mid \iota :=$ *eiterator* $e', \Delta\} =_{df}$ *let* $\iota :=$ *eiterator* e' *in* $\{e \mid \Delta\}$, and $\{e \mid ys :=$ *syncedwith*$(x := e_1, y \in \iota, bf, cs), \Delta\} =_{df}$ *let* $ys :=$ *syncedwith*$(x := e_1, y \in \iota, bf, cs)$ *in* $\{e \mid \Delta\}$.

The following desirable relationship between *syncmap* and *eiterator* and *syncedwith* is not difficult to see.

Theorem 6.5 (Synchrony iterator). *Let X and Y be elists in canonical form. Let $bf(y, x)$ be monotonic with respect to X and Y. Let $cs(y, x)$ be antimonotonic with respect to $bf(y, x)$.*

(1) *syncmap*$(x \in X, y \in ys \subseteq Y, bf, cs, f) =_{df} \{z \mid \iota :=$ *eiterator* $Y, x \in X, ys :=$ *syncedwith*$(x, y \in \iota, bf, cs), z \in f(x, ys)\}$.
(2) *time*(*syncmap*$(x \in X, y \in ys \subseteq Y, bf, cs, f)) =$ *time*$(\{z \mid \iota :=$ *eiterator* $Y, x \in X, ys :=$ *syncedwith*$(x, y \in \iota, bf, cs), z \in f(x, ys)\})$.
(3) $\{z \mid \iota :=$ *eiterator* $Y, x \in X, ys :=$ *syncedwith*$(x, y \in \iota, bf, cs), z \in f(x, ys)\} = \{z \mid x \in X, z \in f(x, \{y \mid y \in Y, cs(y, x)\})\}$.
(4) *$\mathcal{NRC}_1(\leq$, syncmap, sort$)$ and $\mathcal{NRC}_1(\leq$, eiterator, syncedwith, sort$)$ have equal extensional and intensional expressive power. I.e., for any expression $e(\vec{x})$ in $\mathcal{NRC}_1(\leq$, syncmap, sort$)$, there is an expresion $e'(\vec{x})$ in $\mathcal{NRC}_1(\leq$, eiterator, syncedwith, sort$)$ such that $e(\vec{x}) = e'(\vec{x})$ and time$(e(\vec{x})) =$ time$(e'(\vec{x}))$; and vice versa.*

Proof. Let $\mathit{syncmap}(x \in X, y \in ys \subseteq Y, bf, cs, f) \Downarrow C$. Let $X == \{x_1, ..., x_n\}$. Let $C_0 == \{\}$ and $E_0 == Y$. For $1 \leq l \leq n$, let $A_l == \mathit{takewhile}(y \in E_{l-1}, bf(y, x_l)\ \mathit{or}\ cs(y, x_l))$; $B_l == \{u \mid u \in A_l, cs(y, x_l)\}$; $C_l == f(x_l, B_l)$; $D_l == \mathit{dropwhile}(y \in E_{l-1}, bf(y, x_l)\ \mathit{or}\ cs(y, x_l))$; $E_l == B_l \oplus D_l$; and $a_l == (E_l, C_l)$. Then it is easy to see that $C = C_1 \oplus \cdots \oplus C_n$.

Let $\{z \mid \iota := \mathit{eiterator}\ Y, x \in X, ys := \mathit{syncedwith}(x, y \in \iota, bf, cs), z \in f(x, ys)\} \Downarrow C'$. Let $\Gamma_0 = [\iota \mapsto Y]$. For $1 \leq l \leq n$, let $\Gamma_{l-1}, \mathit{syncedwith}(x := x_l, y \in \iota, bf(y, x), cs(y, x)) \Downarrow \Gamma_l, C''_l$; $\Gamma_l(\iota) == \{u_{l,1}, ..., u_{l,m_l}\}$; $bf(u_{l-1,j}, x_l) \Downarrow v_{l,j}$; $cs(u_{l-1,j}, x_l) \Downarrow v'_{l,j}$; $B'_l == \{u_{l-1,j} \mid 1 \leq j < i, v'_{l,j} = \mathit{true}\}$, $D'_l == \{u_{l-1,i}, ..., u_{l-1,m_{l-1}}\}$, and $\Gamma_l(\iota) == B'_l \oplus D'_l$, if $v_{l,j} = \mathit{true}$ or $v'_{l,j} = \mathit{true}$ for $1 \leq j < i$ and $v_{l,i} = \mathit{false}$ and $v'_{l,i} = \mathit{false}$; $B'_l == \{u_{l-1,j} \mid 1 \leq j < m_{l-1}, v'_{l,j} = \mathit{true}\}$, $D'_l == \{\}$, and $\Gamma_l(\iota) == B'_l \oplus D'_l == B'_l$, if $v_{l,j} = \mathit{true}$ or $v'_{l,j} = \mathit{true}$ for $1 \leq j \leq m_{l-1}$; and $f(x_l, B'_l) \Downarrow C'_l$. Since bf, cs, and f do not update the global store, it is easy to see that $C' == C'_1 \oplus \cdots \oplus C'_n$.

Having set the above up, the proof for Part 1 and 2 of the theorem now follows straightforwardly by induction on l that $B'_l == B_l$, $D'_l == D_l$, $C'_l == C_l$, and $\Gamma_l(\iota) == E_l$. For Part 3 of theorem, this follows from Part 1 of this theorem and Part 1 of Theorem 6.3.

Part 1 to 3 of this theorem are readily generalized to $\mathit{syncmap}_2$, $\mathit{syncmap}_3$, etc. For example, $\mathit{syncmap}_2(x \in X, y_1 \in ys_1 \subseteq Y_1, bf_1, cs_1, y_2 \in ys_2 \subseteq Y_2, bf_2, cs_2, f) =_{df} \{z \mid \iota_1 := \mathit{eiterator}\ Y_1, \iota_2 := \mathit{eiterator}\ Y_2, x \in X, ys_1 := \mathit{syncedwith}(x, y_1 \in \iota_1, bf_1, cs_1), ys_2 := \mathit{syncedwith}(x, y_2 \in \iota_2, bf_2, cs_2), z \in f(x, ys_1, ys_2)\}$. Thus, all functions which are expressible in $\mathcal{NRC}_1(\leq, \mathit{syncmap}, \mathit{sort})$ are expressible in $\mathcal{NRC}_1(\leq, \mathit{eiterator}, \mathit{syncedwith}, \mathit{sort})$ at the same time complexity. This settles one direction of Part 4.

For the other direction of Part 4, according to the safe-use constrains, the proof can proceed by regarding bf, cs, and f as pure expressions. By safe-use Constraint 4, the *eiterator* and *syncedwith* constructs always occur in a form like this: $\mathit{let}\ \iota_1 := \mathit{eiterator}\ Y_1\ \mathit{in}\ ...\ \mathit{let}\ \iota_h := \mathit{eiterator}\ Y_h\ \mathit{in} \bigcup\{\ \mathit{let}\ ys_1 := \mathit{syncedwith}(u_1 := x, y_1 \in \iota_1, bf_1, cs_1)\ \mathit{in}\ ...\ \mathit{let}\ ys_h := \mathit{syncedwith}(u_h := x, y_h \in \iota_h, bf_h, cs_h)\ \mathit{in}\ f(x, ys_1, ..., ys_h) \mid x \in X\}$. This can be translated to $\mathit{syncmap}_h(x \in X, y_1 \in ys_1 \subseteq Y_1, bf_1, cs_1, ..., y_h \in ys_h \subseteq Y_h, bf_h, cs_h, f(x, ys_1, ..., ys_h))$. Thus, all functions which are expressible in $\mathcal{NRC}_1(\leq, \mathit{eiterator}, \mathit{syncedwith}, \mathit{sort})$ are also expressible in $\mathcal{NRC}_1(\leq, \mathit{syncmap}, \mathit{sort})$. This settles the other direction of Part 4, and also the theorem. □

Having proved Theorem 6.5, a confession is now in order: The safe-use constraints are actually overly strict. They were imposed earlier to simplify the proof of this theorem. In particular, Constraint 4 can be and should be relaxed to permit expressions of the form

> $\mathit{let}\ \iota_1 := \mathit{eiterator}\ e_1\ \mathit{in}\ ...\ \mathit{let}\ \iota_h := \mathit{eiterator}\ e_h\ \mathit{in} \bigcup\{\ \mathit{if}\ e'(x)\ \mathit{then}\ \mathit{let}\ ys_1 := \mathit{syncedwith}(u_1 := x, y_1 \in \iota_1, bf_1, cs_1)\ \mathit{in}\ ...\ \mathit{let}\ ys_h := \mathit{syncedwith}(u_h := x, y_h \in \iota_h, bf_h, cs_h)\ \mathit{in}\ f(x, ys_1, ..., ys_h)\ \mathit{else}\ \{\} \mid x \in e\}$

In this relaxed form, instead of synchronizing $\iota_1, ..., \iota_h$ to every $x \in e$, they are synchronized to just those $x \in e$ satisfying $e'(x)$. This more relaxed version of Contraint 4 can be derived from the original version via the filter-promotion rule:

> $\mathit{let}\ ys := \mathit{syncedwith}(u := x, y \in \iota, bf, cs)\ \mathit{in}\ \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ \{\} \mapsto$
> $\mathit{if}\ e_1\ \mathit{then}\ \mathit{let}\ ys := \mathit{syncedwith}(u := x, y \in \iota, bf, cs)\ \mathit{in}\ e_2\ \mathit{else}\ \{\}$
> provided ys is not a free variable of e_1

This filter-promotion rule is sound and does not increase time complexity. In fact, it often reduces time complexity.

As shown above, *eiterator* and *syncedwith* work together to define a synchronized iteration on multiple elists. This pair of constructs is hereby called a Synchrony iterator construct: a Synchrony iterator, which one can think of as ι, is introduced by *eiterator* and is used in *syncedwith*. Also, for this reason, *eiterator* actually stands for "enchanced iterator" and is pronounced as "iterator."

6.3 Efficient interval joins

A remarkable byproduct of endowing comprehension syntax with Synchrony iterator is that efficient interval joins with overlap predicates are also straightforward to express as well. This type of joins does not involve any equality test; rather, the join predicates involve testing whether two "intervals" overlap each other. These joins are commonly encountered when processing temporal data (e.g., finding events which overlap in time), genomic data (e.g. finding pairs of transcription factors which have binding sites that are near each other in some gene promoters), and so on.

Relational database systems typically require quadratic time complexity on this type of joins. The reason is that typical relational database systems could only choose merge-join as the execution plan when at least one of the join predicates is an equality test; however, as mentioned, this type of joins has no equality test. For the same reason, relational database systems also could not use an index-based join execution plan even when indices are available, because database indices facilitate fast look-up based on equality test. So, relational database systems are typically left with only nested loops as their execution strategy for this type of joins.

There are specialized algorithms developed in the database community for this kind of joins. For example, Piatov et. al. [18] recently presented an algorithm for interval joins with overlap predicates; the algorithm requires a combination of a new data structure, lazy evaluation technique, and even exploitation of certain features of modern CPU architectures.

So, it is noteworthy that efficient interval joins with overlap predicates come naturally when comprehension syntax is augmented with Synchrony iterator. Below is an example.

Example 6.6. Suppose there are two relations, $X : \mathit{start} \times \mathit{end} \times \mathit{id}$ and $Y : \mathit{start} \times \mathit{end} \times \mathit{id}$. Elements in the two relations can be regarded as intervals on a number line. Suppose also that X and Y are canonical; i.e., they have been sorted in ascending order of the start and end points of the intervals. Let us also write $x.\mathit{start}$, $x.\mathit{end}$, and $x.\mathit{id}$ in place of $x.\pi_1$, $x.\pi_2$, and $x.\pi_3$ for better readability. Let $y\ \mathit{overlap}\ x\ =_{df}\ (x.\mathit{start} < y.\mathit{end} \leq x.\mathit{end})\ \mathit{or}\ (x.\mathit{start} \leq y.\mathit{start} < x.\mathit{end})\ \mathit{or}$ $(y.\mathit{start} < x.\mathit{end} \leq y.\mathit{end})\ \mathit{or}\ (y.\mathit{start} \leq x.\mathit{start} < y.\mathit{end})$, meaning two intervals x and y overlap. As X and Y are canonical, the ordering $<$ on $\mathit{start} \times \mathit{end} \times \mathit{id}$ is trivially monotonic with respective to them. It is also a quick exercise to see that $y\ \mathit{overlap}\ x$ is antimonotonic with respect to $<$.

Consider the following queries:

- $\mathit{join}_4(X, Y) =_{df} \{(x.\mathit{id}, y.\mathit{id}) \mid x \in X, y \in Y, y\ \mathit{overlap}\ x, y.\mathit{end} - y.\mathit{start} < x.\mathit{end} - x.\mathit{start}\}$.
- $\mathit{join}'_4(X, Y) =_{df} \{(x.\mathit{id}, y.\mathit{id}) \mid \iota := \mathit{eiterator}\ Y, x \in X, \mathit{ys} := \mathit{syncedwith}(x, y \in \iota, y < x, y\ \mathit{overlap}\ x), y \in \mathit{ys}, y.\mathit{end} - y.\mathit{start} < x.\mathit{end} - x.\mathit{start}\}$.

Suppose no interval overlaps more than k other intervals. Then, $\mathit{join}_4(X, Y) = \mathit{join}'_4(X, Y)$ and $\mathit{time}(\mathit{join}_4(X, Y)) = \Theta(|X| * |Y|)$, but $\mathit{time}(\mathit{join}'_4(X, Y)) = O(|X| + |Y|)$.

As explained earlier, typical relational database systems would also have time complexity $\Theta(|X| * |Y|)$ on join_4. So, it is gratifying that Synchrony iterator enables comprehension syntax to beat typical relational database systems by achieving linear time complexity $O(|X| + |Y|)$ for join'_4.

A sharp-eyed reader might realise that the above example implies Synchrony iterator has more intensional expressive power than a typical relational database system, since the latter seems to have difficulty realizing efficient interval joins. There is actually some truth in this. Given a join query $f(X, Y) =_{df} \{g(\vec{x}, \vec{y}) \mid \vec{x} \in X, \vec{y} \in Y, p(\vec{x}, \vec{y})\}$. The query optimizer of a typical relational database system tries to decompose this to $f_1(X, Y) =_{df} \{(\vec{x}, \vec{y}) \mid \vec{x} \in X, \vec{y} \in Y, \mathit{cs}(\vec{y}_1, \vec{x}_2)\}$ and $f_2(X, Y) =_{df} \{g(\vec{x}, \vec{y}) \mid (\vec{x}, \vec{y}) \in f_1(X, Y), p'(\vec{x}, \vec{y})\}$, where $p(\vec{x}, \vec{y})$ iff $\mathit{cs}(\vec{y}_1, \vec{x}_1)$ and $p'(\vec{x}, \vec{y})$; $\mathit{cs}(\vec{x}_1, \vec{y}_2)$ is comprised entirely of equality tests; and X is sorted on $\vec{x}_1$ and Y is sorted on $\vec{y}_1$. Then $f_1(X, Y)$ is executed as a merge join. The merge join [2] is essentially an algorithm equivalent to $\mathit{mergejoin}(\mathit{cs}, X, Y) =_{df} \mathit{fold}(a := (Y, \{\}), \vec{x} \in X, (\mathit{dropwhile}(\vec{y} \in a.\pi_1, \vec{y}_1 < \vec{x}_1), a.\pi_2 \cup \{(\vec{x}, \vec{u}) \mid$

$\vec{u} \in takewhile(\vec{y} \in dropwhile(\vec{y} \in a.\pi_1, \vec{y}_1 < \vec{x}_1), cs(\vec{y}_1, \vec{x}_1))\})).\pi_2$; where $cs(\vec{y}_1, \vec{x}_1)$ is a join predicate consisting entirely of equality test. It is not difficult to see that $mergejoin(cs, X, Y) = syncmap(\vec{x} \in X, \vec{y} \in ys \subseteq Y, \vec{y}_1 < \vec{x}_1, cs(\vec{y}_1, \vec{x}_1), \{(\vec{x}, \vec{y}) \mid \vec{y} \in ys\})$. In other words, by restricting the $cs(\vec{y}_1, \vec{x}_1)$ predicate to equality tests, instead of the more general constraint of antimonotonicity, an exact match in intensional expressiveness between Synchrony iterator and a typical relational database system is achieved.

Conversely, if the relational database query optimizer is able to decompose a join predicate $p(\vec{x}, \vec{y})$ into an antimonotonic $cs(\vec{y}_1, \vec{x}_1)$ and a residual $p'(\vec{x}, \vec{y})$, and replace *mergejoin* by *syncmap*, an exact match in intensional expressiveness can also be achieved. The replacement of *mergejoin* by *syncmap* requires only an extremely simple change in one step of the usual merge join algorithm in a relational database system. The merge join [2], in pseudo codes, essentially repeats the following six steps until all of X and Y have been processed. Step 1: Scan a y from Y and an x from X. Step 2: Keep scanning and dropping x until $cs(\vec{y}_1, \vec{x}_1)$ or $\vec{y}_1 < \vec{x}_1$. Step 3: Keep scanning and dropping y until $cs(\vec{y}_1, \vec{x}_1)$ or $\vec{y}_1 > \vec{x}_1$. Step 4: Put this and subsequent y into a work table whenever $cs(\vec{y}_1, \vec{x}_1)$ is true, and as long as $cs(\vec{y}_1, \vec{x}_1)$ remains true. Step 5: Output $(\vec{x}, \vec{y})$ for each y in the work table and each x where $cs(\vec{y}_1, \vec{x}_1)$, until $cs(\vec{y}_1, \vec{x}_1)$ becomes false. Step 6: Empty the work table. To modify this into *syncmap*, it is sufficient to change Step 4 into Step 4': Put this and subsequent y into a work table whenever $cs(\vec{y}_1, \vec{x}_1)$ is true, as long as either $\vec{y}_1 < \vec{x}_1$ or $cs(\vec{y}_1, \vec{x})$ is true.

7 CLOSING REMARKS

The impedance mismatch problem between databases and programming languages has been highlighted three decades ago [7]. It refers to the difficulties of integrating database query-like feature and capability into a programming language. Some has regarded the use of comprehension syntax [3] as a breakthrough for this problem [6]. Indeed, comprehension syntax provides an iteration construct that is simple enough for programming with collection data types that data objects of a database have been mapped to, and explicit enough to admit a direct translation to the query language of the database, thereby permitting queries to the database to be embedded simply and naturally into a programming language.

However, comprehension syntax is also widely adopted in modern programming languages—e.g., Python [10] and Scala [17]— as an easy-to-use means for manipulating collection types in general. For this purpose, the collection objects are created within a program or do not come from a database system, and queries written in comprehension syntax for manipulating these objects are not translated to the query language of an underlying database system for execution. In such a setting, programs written in comprehension syntax typically correspond to nested loops.

This gives rise to an intriguing disparity. Many queries when translated to their database equivalent can be executed by the underlying database system very efficiently. Yet when they are executed directly as comprehension syntax, they are not efficient at all. Consider this query as an example, $\{(x.dept, x.stf) \mid x \in DeptStaff, y \in Staff, x.stf = y.stf, y.age > 65\}$ which retrieves departments and their staff who are above 65 years old. Suppose a staff typically belongs to only one department. This query would then be a low-selectivity join. It typically would be executed by a database system, via e.g. a merge join [2], with time complexity $\Theta(n + m)$ assuming the inputs *DeptStaff* and *Staff* have size n and m and are both sorted by their *stf* field; or with time complexity $\Theta(n \log(n) + m \log(m))$ if sorting is required. In contrast, the same query would typically has time complexity $\Theta(n * m)$ natively in the programming language. Even if a filter promotion is applied (and ignoring the change in the appearance of the output) to optimize the query to $\{(x.dept, x.stf) \mid y \in Staff, y.age > 65, x \in DeptStaff, x.stf = y.stf\}$, this optimized query still has quadratic time complexity $\Theta(g * n * m)$, for some $0 \leq g \leq 1$, natively in the programming language.

This linear-vs-quadratic time complexity difference of low-selectivity joins can be called an intensional expressiveness gap between comprehension syntax and database systems. That is, it is a gap between the algorithms that can be expressed using comprehension syntax and database systems. As far as relational database system is concerned, the low-selectivity join, the relational intersection, and the relational difference appear to be the only intensional expressiveness gap as all other relational query operators, as well as high-selectivity joins, in the absence of database indices on the input relations, have similar time complexity whether executed by a relational database system or in the programming language directly as queries in comprehension syntax.

It has been open whether this intensional expressiveness gap is a real gap; i.e., there might exist some clever way to implement low-selectivity joins efficiently using comprehension syntax. As the first main result of this paper, this intensional expressiveness gap is proved by showing that all subquadratic algorithms expressible using pure comprehension syntax cannot compute low-selectivity joins. In fact, as shown in the second main result of this paper, even allowing some functions—viz. *takewhile* and *dropwhile*, *fold*, or *zip*—commonly available in the collection-type function libraries of programming languages, to be used with comprehension syntax, all expressible subquadratic algorithms still cannot compute low-selectivity joins in general.

It is a natural follow-up question on what exactly is missing from comprehension syntax that prevents efficient algorithms for low-selectivity joins to be expressed. As the third main result of this paper, this intensional expressiveness gap is charaterized in a precise way by identifying a new programming construct to be used with comprehension syntax. This is the Synchrony iterator construct for expressing synchronized iterations on multiple collection objects. A construct for generalized iteration on multiple collection objects in synchrony appears to be a conceptually novel choice, because practically all functions commonly provided in the function libraries of programming languages involve iteration on a single collection object. This Synchrony iterator construct is shown to fill the gap exactly. In particular, adding this construct does not change the functions that are expressible using pure comprehension syntax, and yet enables the realization of efficient low-selectivity joins. Moreover, the Synchrony iterator construct dovetails rather appealingly with comprehension syntax, so that efficient queries written with the help of Synchrony iterators often do not look too different from their inefficient pure comprehension-syntax equivalents.

The proof of the intensional expressiveness gap uses a novel limited-mixing lemma. The lemma shows that all subquadratic-time queries in comprehension syntax are only able to mix atomic objects in their input in very limited ways. This lemma is further extended in the presence of *takewhile* and *dropwhile*, and *zip*, as well as in the presence of *fold*. These limited-mixing lemmas are of independent interest, and constitute the fourth main contribution of this paper. Most past works on intensional expressive power are query specific. Just to cite a couple of examples, Abiteboul and Vianu [1] showed that there is no "generic machine" for computing the parity query in PTIME; and Suciu and Paredaens [19] showed that the transitive closure of a long chain can only be computed in the complex object algebra of Abiteboul and Beeri using exponential space. A notable non-query-specific intensional expressiveness result is that of Wong [25], who showed that all queries on a general class of structures, which includes deep trees and long chains, in a nested relational calculus augmented with a powerset operator are either already expressible in the calculus without using the powerset operator, or must use an exponential amount of space. Furthermore, most previous results on intensional expressive power, such as those mentioned above, are for query languages without ordered data types. The limited-mixing lemmas in this paper stand out in comparison to these results in two aspects. Firstly, the limited-mixing lemmas are non-query specific; they apply to all queries of subquaratic time complexity in the respective query languages. Secondly, the limited-mixing lemmas are valid in the presence of ordered data types. The limited-mixing lemmas thus greatly enrich the repertoire of techniques for studying intensional expressive power. The limited-mixing lemmas are also useful intensional counterparts to Gaifman's locality property [8]. Gaifman's locality property is useful for analyzing the extensional and intensional expressive power [11, 13, 25] of query languages on unordered data types. However, it is effectively useless on ordered data types and on query languages with a *fold*-like function. Limited-mixing lemmas do not have these limitations.

The Synchrony iterator construct $syncedwith(x := e, y \in \iota, bf, cs)$ seems to clarify two concepts required for specifying general algorithms for merging two collections or for synchronized iterations on two collections. The predicate $bf(y, x)$, with its associated monotonicity conditions, intuitively corresponds to the concept of "y is before x." That is, for y and x coming from two collections to be merged, $bf(y, x)$ means y should come before x in the merge result. Hence, $bf(y, x)$ serves as a general bridge for establishing the relative positions of objects in two collections of potentially different types. The predicate $cs(y, x)$, with its associated antimonotonicity conditions, intuitively corresponds to the concept of "y can see x." That is, for y and x coming from two collections to be synchronized, $cs(y, x)$ means a synchronization point has been found and the pair (x, y) should be acted upon. The two antimonotonicity conditions associated with $cs(y, x)$ is worth further highlighting. Given two collections X and Y, where $bf(y, x)$ is monotonic and $cs(y, x)$ is antimonotonic on them; and $x \in X$ and $y \in Y$. The first antimonotonicity condition states that $bf(y, x)$ and not $cs(y, x)$ implies not $cs(y, x')$ for all x' that appear after x in X. This means y can be safely discarded, as every item that it can potentially be synchronized with has already been seen. The second antimonotonicity condition states that not $bf(y, x)$ and not $cs(y, x)$ implies not $cs(y', x)$ for all y' that appear after y in Y. This means x can be safely discarded, as every item that it can potentially be synchronized with has already been seen. Therefore, they can be thought of as a pair of rules for shifting and discarding elements in X and Y as these two collections are being iterated on in synchrony. Such a pair of rules seems intrinsic to merge-like algorithms, and can perhaps serve as a general characterization of what merge-like algorithms are. Indeed, efficient interval joins with overlap predicates, which are not supported well in database systems, are obtained for free using Synchrony iterator as shown earlier.

Finally, here is a small advertisement: Synchrony iterator has been implemented in Python and Scala. These implementations are available at https://www.comp.nus.edu.sg/~wongls/projects/synchrony.

ACKNOWLEDGMENTS

The concept of Synchrony iterators (and Synchrony fold, which is not discussed here) is evolved from a talk that Stefano Ceri invited me to give at the GeCO Workhop on Challenges in Data-Driven Genomic Computing, held in Como, Italy in March 2019. The concept was subsequently refined through discussions with Val Tannen and Stefano Perna. Jeremy Gibbons and several others suggested combinations of *takewhile*, *dropwhile*, and *fold* as possible alternatives to Synchrony iterators. The work on this paper is partially motivated by a desire to clarify the virtues of Synchrony iterators over such alternatives. Incidentally, it was Peter Buneman who conjectured that *zip*, which is a prototypical low-selectivity join, could only be defined using comprehension syntax at quadratic time complexity; this was communicated to me two and half decades ago by Stijn Vansummeren, and apparently have remained open till it is settled here in Proposition 4.2. I am grateful to all of them for their role in the genesis of this work. I am indebted to Jan Van den Bussche, who carefully read a draft of this paper and provided many constructive comments; this paper is much more complete and rigorous due to his effort. I thank also Kian Lee Tan and Chee Yong Chan for highlighting the practical connection for using Synchorny iterator to realize efficient interval joins; Synchrony iterator is made more appealing because of their insight.

This work was supported by National Research Foundation, Singapore, under its Synthetic Biology Research and Development Programme (Award No: SBP-P3); by Ministry of Education, Singapore, Academic Research Fund Tier-1 (Award No: MOE T1 251RES1725); and by a Kwan Im Thong Hood Cho Temple Chair Professorship. Any opinions, findings, and conclusions or recommendations expressed in this material are mine, and do not reflect the views of these grantors.

REFERENCES

[1] Serge Abiteboul and Victor Vianu. 1991. Generic Computation and its Complexity. In *Proceedings of 23rd ACM Symposium on the Theory of Computing*. 209–219.

[2] M. Blasgen and K. Eswaran. 1977. Storage and Access in Relational Databases. *IBM Systems Journal* 16, 4 (1977), 363–377.

[3] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. 1994. Comprehension Syntax. *SIGMOD Record* 23, 1 (March 1994), 87–96.

[4] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science* 149, 1 (September 1995), 3–48.

[5] E. F. Codd. 1972. Relational completeness of data base sublanguages. In *Data Base Systems*, R. Rustin (Ed.). Prentice-Hall, 65–98.

[6] Ezra Cooper. 2009. The script-writer dream: How to write great SQL in your own language, and be sure it will succeed. In *Proceedings of 12th International Symposium on Database Query Languages*. Lyon, France, 36–51.

[7] George Copeland and David Maier. 1984. Making Smalltalk A Database System. In *Proceedings of ACM-SIGMOD 84*. Boston, MA, 316–325.

[8] Haim Gaifman. 1982. On Local and Non-local Properties. In *Proceedings of the Herbrand Symposium, Logic Colloquium '81*. North Holland, 105–135.

[9] Martin Grohe and Thomas Schwentick. 2000. Locality of order-invariant first-order formulas. *ACM Transactions on Computational Logic* 1, 1 (July 2000), 112–130.

[10] John V. Guttag. 2016. *Introduction to Computation and Programming Using Python: With Application to Understanding Data*. MIT Press.

[11] Lauri Hella, Leonid Libkin, and Juha Nurmonen. 1999. Notions of Locality and their Logical Characterizations over Finite Models. *Journal of Symbolic Logic* 64, 4 (1999), 1751–1773.

[12] Dirk Leinders and Jan Van den Bussche. 2007. On the complexity of division and set joins in the relational algebra. *J. Comput. System Sci.* 73, 4 (June 2007), 538–549.

[13] Leonid Libkin. 1997. On the Forms of Locality Over Finite Models. In *Proceedings of 12th IEEE Symposium on Logic in Computer Science*. 204–215.

[14] Leonid Libkin and Limsoon Wong. 1994. Aggregate Functions, Conservative Extension, and Linear Orders. In *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha (Eds.). Springer-Verlag, 282–294. See also UPenn Technical Report MS-CIS-93-36.

[15] Leonid Libkin and Limsoon Wong. 1994. Conservativity of Nested Relational Calculi with Internal Generic Functions. *Inform. Process. Lett.* 49, 6 (March 1994), 273–280.

[16] Leonid Libkin and Limsoon Wong. 1997. Query Languages for Bags and Aggregate Functions. *J. Comput. System Sci.* 55, 2 (October 1997), 241–272.

[17] Martin Odersky, Lex Spoon, and Bill Venners. 2019. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Inc.

[18] Danila Piatov, Sven Helmer, and Anton Dignos. 2016. An interval join optimized for modern hardware. In *Proceedings of 32nd IEEE International Conference on Data Engineering*. 1098–1109.

[19] Dan Suciu and Jan Paredaens. 1997. The complexity of the evaluation of complex algebra expressions. *Journal of Computer and Systems Sciences* 55, 2 (October 1997), 322–343.

[20] Dan Suciu and Limsoon Wong. 1995. On Two Forms of Structural Recursion. In *LNCS 893: Proceedings of 5th International Conference on Database Theory*. Springer-Verlag, Prague, 111–124.

[21] S. J. Thomas and P. C. Fischer. 1986. *Nested Relational Structures*. JAI Press, London, England, 269–307.

[22] W. Wechler. 1992. *Universal Algebra for Computer Scientists*. EATCS Monograph on Theoretical Computer Science, Vol. 25. Springer-Verlag, Berlin.

[23] Limsoon Wong. 1996. Normal Forms and Conservative Extension Properties for Query Languages over Collection Types. *J. Comput. System Sci.* 52, 3 (June 1996), 495–505.

[24] Limsoon Wong. 2000. Kleisli, a Functional Query System. *Journal of Functional Programming* 10, 1 (2000), 19–56.

[25] Limsoon Wong. 2013. A dichotomy in the intensional expressive power of nested relational calculi augmented with aggregate functions and a powerset operator. In *Proceedings of 32nd ACM Symposium on Principles of Database Systems*.

New York, 285–295.