JFP, **??** pages, 2020. © Cambridge University Press 2020
doi:10.1017/xxxxx

Iterating on multiple collections in synchrony

STEFANO PERNA

Department of Computer Science,
National University of Singapore
(e-mail: dcsstef@nus.edu.sg)

VAL TANNEN

Department of Computer and Information Science,
University of Pennsylvania
(e-mail: val@cis.upenn.edu)

LIMSOON WONG

Department of Computer Science,
National University of Singapore
(e-mail: wongls@comp.nus.edu.sg)

Abstract

Modern programming languages typically provide some form of comprehension syntax which renders programs manipulating collection types more readable and understandable. However, comprehension syntax corresponds to nested loops in general. There is no simple way of using it to express efficient general synchronized iterations on multiple ordered collections, such as linear-time algorithms for low-selectivity database joins. *Synchrony fold* is proposed here as a novel characterization of synchronized iteration. Central to this characterization is a *monotonic* `isBefore` *predicate* for relating the orderings on the two collections being iterated on, and an *antimonotonic* `canSee` *predicate* for identifying matching pairs in the two collections to synchronize and act on.

A restriction is then placed on Synchrony fold, cutting its extensional expressive power to match that of comprehension syntax, giving us *Synchrony generator*. Synchrony generator retains sufficient intensional expressive power for expressing efficient synchronized iteration on ordered collections. In particular, it is proved to be a natural generalization of the database merge join algorithm, extending the latter to more general database joins. Finally, *Synchrony iterator* is derived from Synchrony generator as a novel form of iterator. While Synchrony iterator has the same extensional and intensional expressive power as Synchrony generator, the former is better dovetailed with comprehension syntax. Thereby, algorithms requiring synchronized iterations on multiple ordered collections, including those for efficient general database joins become expressible naturally in comprehension syntax.

1 Introduction

Comprehension syntax, together with simple appeals to library functions, usually provides clear, understandable and short programs. Such a programming style for collection manipulation avoids loops and recursion as these are regarded as harder to understand and more error-prone. However, current collection-type function libraries appear lacking direct support that takes effective advantage of a *linear ordering* on collections for programming

in the comprehension style, even when such an ordering can often be made available by sorting the collections at a linearithmic overhead. We do not argue that these libraries lack expressive power *extensionally*, as the *functions* that interest us on ordered collections are easily expressible in the comprehension style. However, they are expressed *inefficiently* in such a style. We give a practical example in Section 2. We sketch in Section 3 proofs that, under a suitable formal definition of the restriction that gives the comprehension style, efficient algorithms for low-selectivity database joins, for example, cannot be expressed. Moreover, in this setting, these algorithms remain inexpressible even when access is given to any single library function such as `foldLeft`, `takeWhile`, `dropWhile`, and `zip`.

We proceed to fill this gap through several results on the design of a suitable collection-type function. We notice that most functions in these libraries are defined on one collection. There is no notion of any form of general synchronized traversal of two or more collections other than `zip`-like mechanical lock-step traversal. This seems like a design gap: synchronized traversals are often needed in real-life applications and, for an average programmer, efficient synchronized traversals can be hard to implement correctly.

Intuitively, a "synchronized traversal" of two collections is an iteration on two collections where the "moves" on the two collections are coordinated, so that the current position in one collection is not too far from the current position in the other collection; i.e., from the current position in one collection, one "can see" the current position in the other collection. However, defining the idea of "position" based on physical position, as in `zip`, seems restrictive. So, a more flexible notion of position is desirable. A natural and logical choice is that of a linear ordering relating items in the two collections; i.e. a linear ordering on the union of the two ordered collections. Also, given two collections which are sorted according to the linear orderings on their respective items, a reasonable new linear ordering on the union should respect the two linear orderings on the two original collections; i.e. given two items in an original collection where the first "is before" the second in the original collection, then the first should be before the second in the linear ordering defined on the union of the two collections.

Combining the two motivations above, our main approach to reducing the complexity of the expressed algorithms is to traverse two or more sorted collections in a *synchronized* manner, taking advantage of relationships between the linear orders on these collections. The following summarizes our results.

An addition to the design of collection-type function libraries is proposed in Section 4. It is called *Synchrony fold*. Some theoretical conditions, viz. monotonicity and antimonotonicity, that characterize efficient synchronized iteration on a pair of ordered collections are presented. These conditions ensure the correct use of Synchrony fold. Synchrony fold is then shown to address the intensional expressive power gap articulated above.

Synchrony fold has the same extensional expressive power as `foldLeft`; it thus captures functions expressible by comprehension syntax augmented with typical collection-type function libraries. Because of this, Synchrony fold is not sufficiently precisely filling the intensional expressive power gap for comprehension syntax sans library function. A restriction to Synchrony fold is proposed in Section 5. This restricted form is called *Synchrony generator*. It has exactly the same extensional expressive power as comprehension syntax without any library function, but it has the intensional expressive power to express efficient

algorithms for low-selectivity database joins. Synchrony generator is further shown to correspond to a rather natural generalization of the database merge join algorithm (Blasgen & Eswaran, 1977; Mishra & Eich, 1992). The merge join was proposed half a century ago, and has remained as a backbone algorithm in modern database systems for processing equijoin and some limited form of non-equijoin (Silberschatz *et al.*, 2016), especially when the result has to be outputted in a specified order. Synchrony generator generalizes it to the class of non-equijoin whose join predicate satisfies certain antimonotonicity conditions.

Previous works have proposed alternative ways for compiling comprehension syntax, to enrich the repertoire of algorithms expressible in the comprehension style. For example, Wadler & Peyton Jones (2007) and Gibbons (2016), have enabled many relational database queries to be expressed efficiently under these refinements to comprehension syntax. However, these refinements only took equijoin into consideration; non-equijoin remains inefficient in comprehension syntax. In view of this and other issues, an iterator form—called *Synchrony iterator*—is derived from Synchrony generator in Section 6. While Synchrony iterator has the same extensional and intensional expressive power as Synchrony generator, it is more suitable for use in synergy with comprehension syntax. Specifically, Synchrony iterator makes efficient algorithms for simultaneous synchronized iteration on multiple ordered collections expressible in comprehension syntax.

Last but not least, Synchrony fold, Synchrony generator, and Synchrony iterator have an additional merit compared to other codes for synchronized traversal of multiple sorted collections. Specifically, they decompose such synchronized iterations into three orthogonal aspects, viz. relating the ordering on the two collections, identifying matching pairs, and acting on matching pairs. This orthogonality arguably makes for a more concise and precise understanding and specification of programs, hence improved reliability, as articulated by Schmidt (1986), Sebesta (2010) and Hunt & Thomas (2000).

2 Motivating example

Let us first define events as a data type.[1] An event has a `start` and an `end` point, where `start` $<$ `end`, and typically has some additional attributes (e.g., an `id`) which do not concern us for now; cf. Figure 1. Events are ordered lexicographically by their `start` and `end` point: If an event `y` starts before an event `x`, then the event `y` is ordered before the event `x`; and when both events start together, the event which ends earlier is ordered before the event which ends later. Some predicates can be defined on events; e.g., in Figure 1, `isBefore(y, x)` says event `y` is ordered before event `x`, and `overlap(y, x)` says events `y` and `x` overlap each other.

Consider two collections of events, `xs: Vec[Event]` and `ys: Vec[Event]`, where `Vec[·]` denotes a generic collection type, e.g., a vector.[2] The function `ov1(xs, ys)`, defined in

[1] Scala (Odersky *et al.*, 2019) is used in this paper as the ambient language for a concrete discussion.

[2] In this paper, for convenience, `Vec[·]` is taken as the Scala `Vector[·]`. This allows us to assume postpends `:+` and `:++` are constant/linear time in their right argument, and preprends `+:` and `++:` are constant/linear time in their left argument. It is fine to take `Vec[·]` as `List[·]`; in this case, the postpends should be swapped by prepends, and some `reverse` has to be inserted into some of the codes. These list-specific details are not germane to understanding the key ideas of this paper. Hence, we adopt vectors as our generic collection type in general. Nonetheless, when we reach the final description of our last result, Synchrony iterator, at the end of Section 6.1, we will use concrete collection types, including an instance of list.

```
case class Event(start: Int, end: Int, id: String)
// Constraint: start < end

val isBefore = (y: Event, x: Event) => {
  (y.start < x.start) ||
  (y.start == x.start && y.end < x.end)
}

val overlap = (y: Event, x: Event) => {
  (x.start < y.end && y.start < x.end)
}


def ov1(xs: Vec[Event], ys: Vec[Event]) = {
  for (x <- xs; y <- ys; if overlap(y, x)) yield (x, y)
}


def ov2(xs: Vec[Event], ys: Vec[Event]) = {
  // Requires: xs and ys sorted lexicographically by (start, end).
  def aux(
    xs: Vec[Event], ys: Vec[Event],
    zs: Vec[Event], acc: Vec[(Event, Event)])
  : Vec[(Event, Event)] =
    // Key Invariant: aux(xs, ys, Vec(), acc) = acc ++ ov1(xs, ys)
    if (xs.isEmpty) acc
    else if (ys.isEmpty && zs.isEmpty) acc
    else if (ys.isEmpty) aux(xs.tail, zs, Vec(), acc)
    else {
      val (x, y) = (xs.head, ys.head)
      (isBefore(y, x), overlap(y, x)) match {
        case (true, false)  => aux(xs, ys.tail, zs, acc)
        case (false, false) => aux(xs.tail, zs ++: ys, Vec(), acc)
        case (_, true)      => aux(xs, ys.tail, zs :+ y, acc :+ (x, y))
        }
      }
  aux(xs, ys, Vec(), Vec())
}
```

Fig. 1. A motivating example. The funtions `ov1(xs, ys)` and `ov2(xs, ys)` are equal on inputs `xs` and `ys` which are sorted lexicographically by their `start` and `end` point. While `ov1(xs, ys)` has quadratic time complexity $O(|\texttt{xs}| \cdot |\texttt{ys}|)$, `ov2(xs, ys)` has time complexity $O(|\texttt{xs}| + k|\texttt{ys}|)$ when each event in `ys` overlaps fewer than k events in `xs`.

Figure 1, retrieves the events in `xs` and `ys` that overlap each other. Although this comprehension syntax-based definition has the important virtue of being clear and succinct, it has quadratic time complexity $O(|\texttt{xs}| \cdot |\texttt{ys}|)$. An alternative implementation `ov2(xs, ys)` is given in Figure 1 as well. On `xs` and `ys` which are sorted lexicographically by `(start, end)`, `ov1(xs, ys)` = `ov2(xs, ys)`. Notably, the time complexity of `ov2(xs, ys)` is $O(|\texttt{xs}| + k|\texttt{ys}|)$, provided each event in `ys` overlaps fewer than k events in `xs`. The proofs for these claims will become obvious later, from Theorem 4.4.

The function `ov1(xs, ys)` exemplifies a database join, and the join predicate is `overlap(y, x)`. Joins are ubiquitous in database queries. Sometimes, a join predicate is a conjunction of equality tests; this is called an equijoin. However, when a join predicate comprises entirely of inequality tests, it is called a non-equijoin; `overlap(y, x)` is a special form of non-equijoin which is sometimes called an interval join. Non-equijoin is quite common in practical applications. For example, given a database of taxpayers, a query

retrieving all pairs of taxpayers where the first earns more but pays less tax than the second is an interval join. As another example, given a database of mobile phones and their prices, a query retrieving all pairs of competing phone models (i.e. the two phone models in a pair are priced close to each other) is another special form of non-equijoin called a band join.

Returning to `ov1(xs, ys)` and `ov2(xs, ys)`, the upper bound k on the number of events in `xs` that an event in `ys` can overlap with is called the selectivity of the join.[3] When restricted to `xs` and `ys` which are sorted by their `start` and `end` point, `ov1(xs, ys)` and `ov2(xs, ys)` define the same function. However, their time complexity is completely different. The time complexity of `ov1(xs, ys)` is quadratic. On the other hand, the time complexity of `ov2(xs, ys)` is a continuum from linear to quadratic, depending on the selectivity k. In a real-life database query, k is often a very small number, relative to the number of entries being processed. So, in practice, `ov2(xs, ys)` is linear.

The definition `ov1(xs, ys)` has the advantage of being obviously correct, due to its being expressed using easy-to-understand comprehension syntax. Whereas, `ov2(xs, ys)` is likely to take even a skilled programmer much more effort to get right. This example is one of many functions having the following two characteristics. Firstly, these functions are easily expressible in a modern programming language using only comprehension syntax. However, this usually results in a quadratic or higher time complexity. Secondly, there are linear-time algorithms for these functions. Yet, there is no straightforward way to provide linear-time implementation for these functions using comprehension syntax without using more sophisticated features of the programming language and its collection-type libraries.

The proof for this intensional expressiveness gap in a simplified theoretical setting is outlined in the next section and is shown in full in a companion paper (Wong, 2021). It is the main objective of this paper to fill this gap as simply as possible.

3 Intensional expressiveness gap

As alluded to earlier, what we call *extensional expressive power* in this paper refers to the class of mappings from input to output that can be expressed, as in Fortune *et al.* (1983) and Felleisen (1991). In particular, so long as two programs in a language $\mathscr{L}$ produce the same output given the same input, even when these two programs differ greatly in terms of time complexity, they are regarded as expressing (implementing) the same function f, and are thus equivalent and mutually substitutable.

However, we focus here on improving the ability to express algorithms, that is, on *intensional expressive power*. Specifically, as in many past works (Abiteboul & Vianu, 1991; Biskup *et al.*, 2004; Van den Bussche, 2001; Colson, 1991; Suciu & Paredaens, 1997; Suciu & Wong, 1995; Wong, 2013), we approach this in a coarse-grained manner by considering the time complexity of programs. In particular, an algorithm which implements a function f in $\mathscr{L}$ is considered inexpressible in a specific setting if every program implementing f in $\mathscr{L}$ under that setting has a time complexity higher than this algorithm.

Since Scala and other general programming languages are Turing complete, in order to capture the class of programs that we want to study with greater clarity, a restriction

[3] The results in this work remain valid when k is defined instead as the average number (rounded up to a whole number) of events in `xs` that an event in `ys` overlaps with.

needs to be imposed. Informally, user-programmers[4] are allowed to use comprehension syntax, collections, and tuples; but they are not allowed to use while-loops, recursion, and nested collections; and they are not allowed to define new data types and new higher-order functions (a higher-order function is a function whose result is another function or is a nested collection.) They are also not allowed to call functions in the collection-type function libraries of these programming languages, unless specifically permitted.

This is called the "first-order restriction." Under this restriction, following Suciu & Wong (1995), when a user-programmer is allowed to use a higher-order function from a collection-type library, e.g., `foldLeft(e)(f)`, the function `f` which is user-defined can only be a first-order function. A way to think about this restriction is to treat higher-order library functions as a part of the syntax of the language, rather than as higher-order functions.

Under such a restriction, some functions may become inexpressible; and even when a function is expressible, its expression may correspond to a drastically inefficient algorithm (Biskup *et al.*, 2004; Suciu & Paredaens, 1997; Wong, 2013). In terms of extensional expressive power, the first-order restriction of our ambient language, Scala, is easily seen to be complete with respect to flat relational queries (Buneman *et al.*, 1995; Libkin & Wong, 1997), which are queries that a relational database system supports (such as joins). The situation is less clear from the perspective of intensional expressive power.

This section outlines results suggesting that Scala under the first-order restriction cannot express efficient algorithms for low-selectivity joins, and that this remains so even when a programmer is permitted to access some functions in Scala's collection-type libraries. Formal proofs are provided in a companion paper (Wong, 2021).

To capture the first-order restriction on Scala, consider the nested relational calculus $\mathscr{NRC}$ of Wong (1996). $\mathscr{NRC}$ is a simply-typed lambda calculus with Boolean and tuple types and their usual associated operations; set types, and primitives for empty set, forming singleton sets, union of sets, and `flatMap` for iterating on sets;[5] and base types with equality tests and comparison tests. Replace its set type with linearly ordered set types, and assume that ordered sets, for computational complexity purposes, are traversed in order in linear time; this way, ordered sets can be thought of as lists. The replacement of set types by linearly ordered set types does not change the nature of $\mathscr{NRC}$ in any drastic way, because $\mathscr{NRC}$ can express a linear ordering on any arbitrarily deeply nested combinations of tuple and set types given any linear orderings on base types; cf. Libkin & Wong (1994). Next, restrict the language to its flat fragment; i.e., nested sets are not allowed. This restriction has no impact on the extensional expressive power of $\mathscr{NRC}$ with respect to functions on non-nested sets, as shown by Wong (1996). Denote this language as $\mathscr{NRC}_1(\leq)$, where the permitted extra primitives are listed explicitly between the brackets.

[4] In this paper, we separate an implementer-programmer who implements programming constructs and library functions from a user-programmer who uses these. The former has access to all features of the programming language. The latter, in the context of this paper, is restricted to Scala under the first-order constraint plus specifically permitted library functions which the former provides. When we say a programmer, we refer to either programmer. So, in this paper, the implementer-programmer is the one implementing the proposed Synchrony fold, Synchrony generator, and Synchrony iterator. And the user-programmer is the one implementing the examples `ovi`, `ovCount`, `mtgi`, etc.

[5] `FlatMap` is Scala's terminology. It is also known as `bind` in the Haskell parlance.

Some terminologies are needed for stating the results. To begin, by an object, we mean the value of any combination of base types, tuples, and sets that is constructible in $\mathscr{NRC}_1(\leq)$.

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 set 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 a set. The notations $atom^0(C)$, $atom^1(C)$, and $atom^{\leq 1}(C)$ 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 sets in C. The notation $molecule^0(C)$ 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 $atom^0(C) = \{c_1, c_2\}$, $atom^1(C) = \{c_3, c_4, c_5, c_6\}$, $atom^{\leq 1}(C) = \{c_1, c_2, c_3, c_4, c_5, c_6\}$, and $molecule^0(C) = \{\{(c_3, c_4), (c_5, c_6)\}\}$.

The level-0 Gaifman graph of an object C is defined as an undirected graph $gaifman^0(C)$ 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 $gaifman^1(C)$ 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 $atom^0(C_i)$ for some $1 \leq i \leq n$; if $C = (C_1, ..., C_n)$, the edges are pairs $(x, y) \in gaifman^1(C_i)$ for some $1 \leq i \leq n$; and there are no other edges. The Gaifman graph of an object C is defined as $gaifman(C) = gaifman^0(C) \cup gaifman^1(C)$; cf. Gaifman (1982).

Let $e(\vec{X})$ be an expression e whose free variables are $\vec{X}$. Let $e[\vec{C}/\vec{X}]$ denote the closed expression obtained by replacing the free variables $\vec{X}$ by the corresponding objects $\vec{C}$. Let $e[\vec{C}/\vec{X}] \Downarrow C'$ mean the closed expression $e[\vec{C}/\vec{X}]$ evaluates to the object C'; the evaluation is performed according to a typical call-by-value operational semantics (Wong, 2021).

It is shown by Wong (2021) that $\mathscr{NRC}_1(\leq)$ expressions can only manipulate their input in highly restricted local manners. In particular, expressions which have at most 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.

Lemma 3.1 (Wong (2021), Lemma 3.1)**.** *Let $e(\vec{X})$ be an expression in $\mathscr{NRC}_1(\leq)$. Let objects $\vec{C}$ have the same types as $\vec{X}$, and $e[\vec{C}/\vec{X}] \Downarrow C'$. Suppose $e(\vec{X})$ has at most linear time complexity with respect to the size of $\vec{X}$.*[6] *Then 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 $v \in atom^0(\vec{C})$.*

Here is a grossly simplified informal argument to provide some insight on this "limited-mixing" lemma. Consider an expression X `flatMap` f, where X is the variable representing the input collection and f is a function to be performed on each element of X in the usual manner of `flatMap`. Then, the time complexity of this expression is $O(n \cdot \hat{f})$, where n is the number of items in X and $O(\hat{f})$ is the time complexity of f. Clearly, $O(n \cdot \hat{f})$ can be linear only when $O(\hat{f}) = O(1)$. Intuitively, this means f cannot have a subexpression of the form X `flatMap` g. Since `flatMap` is the sole construct in $\mathscr{NRC}_1(\leq)$ for accessing and manipulating the elements of a collection, when f is passed an element of X, there is no way for it to access a different element of X if f does not have a subexpression of the form X `flatMap` g. So, it is not possible for f to mix the components from two different elements

[6] In this work, all mentions of time complexity are with respect to input size.

of X. Unavoidably, many details are swept under the carpet in this informal argument, but are taken care of by Wong (2021).

This limited-mixing handicap remains when the language is further augmented with typical functions—such as `dropWhile`, `takeWhile`, and `foldLeft`—in collection-type libraries of modern programming languages. Even the presence of a fictitious operator, `sort`, for instantaneous sorting, cannot rescue the language from this handicap.

Lemma 3.2 (Wong (2021), Lemma 5.1)**.** *Let $e(\vec{X})$ be an expression in $\mathscr{NRC}_1(\leq$, `takeWhile`, `dropWhile`, `sort`$)$. Let objects $\vec{C}$ have the same types as $\vec{X}$, and $e(\vec{C}/\vec{X}) \Downarrow C'$. Suppose $e(\vec{X})$ has at most 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 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 $v \in atom^1(\vec{C})$, $u \in atom^1(\vec{C})$ and $v \in A$.*

Lemma 3.3 (Wong (2021), Lemma 5.4)**.** *Let $e(\vec{X})$ be an expression in $\mathscr{NRC}_1(\leq$, `foldLeft`, `sort`$)$. Let objects $\vec{C}$ have the same types as $\vec{X}$, and $e(\vec{C}/\vec{X}) \Downarrow C'$. Suppose $e(\vec{X})$ has at most 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 $|A| \leq k$, such 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 $u \in atom^1(\vec{C})$ and $v \in atom^0(\vec{C})$, or $u \in A$ and $v \in atom^1(\vec{C})$, or $v \in A$ and $u \in atom^1(\vec{C})$.*

The inexpressibility of efficient algorithms for low-selectivity joins in $\mathscr{NRC}_1(\leq)$, $\mathscr{NRC}_1(\leq$, `takeWhile`, `dropWhile`, `sort`$)$, and $\mathscr{NRC}_1(\leq$, `foldLeft`, `sort`$)$ can be deduced from Lemmas 3.1, 3.2, and 3.3. The argument for $\mathscr{NRC}_1(\leq$, `foldLeft`, `sort`$)$ is provided here as an illustration. Let `zip(xs, ys)` be the query that pairs the ith element in `xs` with the ith element in `ys`, assuming that the two input collections `xs` and `ys` are sorted and have the same length. Without loss of generality, suppose the ith element in `xs` has the form (o_i, x_i) and that in `ys` has the form (o_i, y_i); suppose also that each o_i occurs only once in `xs` and once in `ys`, each x_i does not appear in `ys`, and each y_i does not appear in `xs`. Clearly, `zip(xs, ys)` is a low-selectivity join; in fact, its selectivity is precisely one. Let $\texttt{xs} = \{(o_1,u_1), ..., (o_n,u_n)\}$ and $\texttt{ys} = \{(o_1,v_1), ..., (o_n,v_n)\}$. Let $C' = \{(u_1,o_1,o_1,v_1), ..., (u_n,o_n,o_n,v_n)\}$. Then $\texttt{zip(xs, ys)} = C'$. Then $gaifman(C') = \{(u_1,v_1), ..., (u_n,v_n)\} \cup \Delta$, where Δ are the edges involving the o_j's in $gaifman(C')$. Clearly, for $1 \leq i \leq n$, $(u_i,v_i) \in gaifman(C')$ but $(u_i,v_i) \notin gaifman(\texttt{xs},\texttt{ys}) = \texttt{xs} \cup \texttt{ys}$. Now, for a contradiction, suppose $\mathscr{NRC}_1(\leq$, `foldLeft`, `sort`$)$ has a linear-time implementation for `zip`. Then, by Lemma 3.3, either $u_i \in atom^0(\texttt{xs},\texttt{ys})$, or $v_i \in atom^0(\texttt{xs},\texttt{ys})$, or $u_i \in A$, or $v_i \in A$ for some A whose size is independent of `xs` and `ys`. However, `xs` and `ys` are both sets; thus, $atom^o(\texttt{xs},\texttt{ys}) = \{\}$. This means A has to contain every u_i or v_i. So, $|A| \geq n = |\texttt{xs}| = |\texttt{ys}|$ cannot be independent of `xs` and `ys`. This contradiction implies there is no linear-time implementation of `zip` in $\mathscr{NRC}_1(\leq$, `foldLeft`, `sort`$)$.

A careful reader may realise that $\mathscr{NRC}_1(\leq)$ does not have the `head` and `tail` primitives commonly provided for collection types in programming languages. However, the absence of `head` and `tail` in $\mathscr{NRC}_1(\leq)$ is irrelevant in the context of this paper. To see this, consider these two functions: $\texttt{take}_n\texttt{(xs)}$ which returns in $O(n)$ time the first n elements of `xs`, and $\texttt{drop}_n\texttt{(xs)}$ which drops in $O(n)$ time the first n elements of `xs`, when `xs` is ordered.

So, `head(xs)` $=$ `take`$_1$`(xs)` and `tail(xs)` $=$ `drop`$_1$`(xs)`. The proof given by Wong (2021) for Lemma 3.2 can be copied almost verbatim to obtain an analogous limited-mixing result for $\mathscr{NRC}_1(\leq, \texttt{take}_n, \texttt{drop}_n, \texttt{sort})$.

Since `zip` is a manifestation of the intensional expressive power gap of $\mathscr{NRC}_1(\leq)$ and its extensions above, one might try to augment the language with `zip` as a primitive. This makes it trivial to supply an efficient implementation of `zip`. Unfortunately, this does not escape the limited-mixing handicap either.

Lemma 3.4 (Wong (2021), Lemma 5.7)**.** *Let $e(\vec{X})$ be an expression in $\mathscr{NRC}_1(\leq,$ `zip`, `sort`$)$. Let objects $\vec{C}$ have the same types as $\vec{X}$, and $e(\vec{C}/\vec{X}) \Downarrow C'$. Suppose $e(\vec{X})$ has at most 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 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 $v \in atom^0(\vec{C})$.*

It follows from Lemma 3.4 that there is no linear-time implementation of `ov1(xs, ys)` in $\mathscr{NRC}_1(\leq,$ `zip`, `sort`$)$. To see this, suppose for a contradiction that there is an expression $f($`xs`, `ys`$)$ in $\mathscr{NRC}_1(\leq,$ `zip`, `sort`$)$ that implements `ov1(xs, ys)` with time complexity $O(|\texttt{xs}| + h|\texttt{ys}|)$ when each event in `ys` overlaps fewer than h events in `xs`. Let k_0 be the k induced by Lemma 3.4 on f. Suppose without loss of generality that no start and end points in `xs` appears in `ys`, and vice versa. Then setting $h > k_0$ produces the desired contradiction.

$\mathscr{NRC}_1(\leq)$ is designed to express the same functions and algorithms that first-order restricted Scala is able to express. A bare-bone fragment of Scala that corresponds to $\mathscr{NRC}_1(\leq)$ can be described as follows. In terms of data types: Base types such as `Boolean`, `Int`, and `String` are included. The operators on base types are restricted to $=$ and $\leq$ tests. Other operators on base types (e.g., functions from base types to base types) can generally be included without affecting the limited-mixing lemmas. Tuple types over base types (i.e., all tuple components are base types) are included. The operators on tuple types are the tuple constructor and the tuple projection. A collection type is included, and the Scala `Vector[·]` is a convenient choice as a generic collection type; however, only vectors of base types and vectors of tuples of base types are included. The operators on vectors are the vector constructor, the `flatMap` on vectors, the vector append `++`, and the vector emptiness test; when restricted to these operators, vectors essentially behave as sets. It is also possible to use other Scala collection types—e.g., `List[·]`—instead of `Vector[·]`, so long as the operators are restricted to a constructor, append `++`, and emptiness test. Some other common operators on collection types, e.g., `head` and `tail`, can also be included, though adding these would make the language correspond to $\mathscr{NRC}_1(\leq, \texttt{take}_1, \texttt{drop}_1)$ instead of $\mathscr{NRC}_1(\leq)$ and, as explained earlier, this does not impact the limited-mixing lemmas. In terms of general programming constructs: Defining functions whose return types are any of the data types above (i.e. return types are not allowed to be function types), making function calls, and using comprehension syntax and if-then-else are all permitted. Although pared to such a bare bone, this highly restricted form of Scala retains sufficient expressive power; e.g., all flat relational queries can be easily expressed using it.

Thus, Lemma 3.1 implies there is no efficient implementation of low-selectivity joins, including `ov1(xs, ys)`, in first-order restricted Scala. Lemma 3.2 implies there is no efficient implementation of low-selectivity joins in first-order restricted Scala even when the programmer is given access to `takeWhile` and `dropWhile`. Lemma 3.3 implies there is no efficient implementation of low-selectivity joins in first-order restricted Scala even when the programmer is given access to `foldLeft`. Lemma 3.4 implies there is no efficient implementation of low-selectivity joins in first-order restricted Scala even when the programmer is given access to `zip`. Moreover, these limitations remain even when the programmer is further given the magical ability to do sorting infinitely fast.

4 Synchrony fold

Comprehension syntax is typically translated into nested `flatMap`'s, each `flatMap` iterating independently on a single collection. Consequently, like any function defined using comprehension syntax, the function `ov1(xs, ys)` in Figure 1 is forced to use nested loops to process its input. While it is able to return correct results even for an unsorted input, it overkills and overpays a price in its quadratic time complexity when its input is already appropriately sorted. In fact, `ov1(xs, ys)` is still overpaying the quadratic-time complexity price when its input is unsorted, because sorting can always be performed when needed for a relatively affordable linearithmic overhead.

In contrast, the function `ov2(xs, ys)` in Figure 1 is linear in time complexity when selectivity is low, which is much more efficient than `ov1(xs, ys)`. There is one fundamental explanation for this efficiency: The input `xs` and `ys` are sorted and `ov2(xs, ys)` directly exploits this sortedness to iterate on `xs` and `ys` in "synchrony," i.e. in a coordinated manner, akin to the merge step in a merge sort (Knuth, 1973) or a merge join (Blasgen & Eswaran, 1977; Mishra & Eich, 1992). However, its codes are harder to understand and to get right.

It is desirable to have an easy-to-understand-and-check linear-time implementation that is as efficient as `ov2(xs, ys)` but using only comprehension syntax, without the acrobatics of recursive functions, while-loops, etc. This leads us to the concepts of *Synchrony fold*, *Synchrony generator*, and *Synchrony iterator*. Synchrony fold is presented in this section. Synchrony generator and iterator are presented later in Sections 5 and 6 respectively.

4.1 Theory of Synchrony fold

The function `ov2(xs, ys)` exploits the sortedness and the relationship between the orderings of `xs` and `ys`. In Scala's collection-type function libraries, functions such as `foldLeft` are also able to exploit the sortedness of their input. Yet there is no way of individually using `foldLeft` and other collection-type library functions mentioned earlier—as suggested by Lemma 3.2, Lemma 3.3, and Lemma 3.4—to obtain linear-time implementation of low-selectivity joins, without defining recursive functions, while-loops, etc. The main reason is that these library functions are mostly defined on a single input collection. Hence, it is hard for them to exploit the relationship between the orderings on two collections. And there is no obvious way to process two collections using any one of these library functions alone, other than in a nested-loop manner, unless the ambient programming language has more

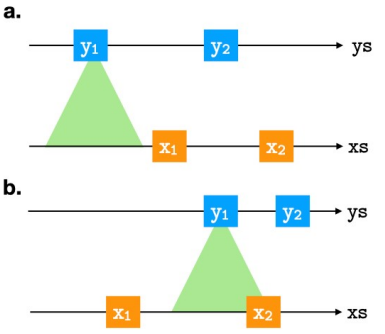

Fig. 2. Visualization of monotonicity and antimonotonicity. Two collections `xs` and `ys` are sorted according to some orderings, as denoted by the two arrows. The `isBefore` predicate is represented by the relative horizontal positions of items x_i and y_j; i.e., if y_j has a horizontal position to the left of x_i, then y_j is before x_i. The `canSee` predicate is represented by the shaded green areas. **a.** If y_1 is before x_1 and cannot see x_1, then y_1 is also before and cannot see any x_2 which comes after x_1. So, every x_i that matches y_1 has been seen; it is safe to move forward to y_2. **b.** If y_1 is not before x_1 and cannot see x_1, then any y_2 which comes after y_1 is also not before and cannot see x_1. So, every y_j that matches x_1 has been seen; it is safe to move forward to x_2.

sophisticated ways to compile comprehensions (Marlow *et al.*, 2016; Wadler & Peyton Jones, 2007), or unless multiple library functions are used together.

Scala's collection-type libraries do provide the function `zip` which pairs up elements of two collections according to their physical position in the two collections, viz. first with first, second with second, and so on. However, by Lemma 3.4, this mechanical pairing by `zip` cannot be used to implement efficient low-selectivity joins, which require more general notions of pairing where pairs can form from different positions in the two collections.

So, we propose `syncFold`, a generalization of `foldLeft` that iterates on two collections in a more flexible and synchronized manner. For this, we need to relate positions in two collections by introducing two logical predicates `isBefore(y, x)` and `canSee(y, x)`, which are supplied to `syncFold` as two of its arguments. Informally, `isBefore(y, x)` means that, when we are iterating on two collections `xs` and `ys`, in a synchronized manner, we should encounter the item `y` in `ys` before we encounter the item `x` in `xs`. And `canSee(y, x)` means that the item `y` in `ys` corresponds to or matches the item `x` in `xs`; in other words, `x` and `y` form a pair which is of interest. Note that an item `y` "corresponds to or matches" an item `x` does not necessarily mean the two items are the same. For example, when items are events as defined in Section 2, in the context of `ov1` and `ov2`, an event `y` corresponds to or matches an event `x` means the two events overlap each other. Obviously, an item does not need to be an atomic object; it can be a tuple or an object having a more complex type.

The `isBefore(y, z)` and `canSee(y, x)` predicates are characterized respectively by the monotonicity and antimonotonicity conditions defined below and depicted in Figure 2. To provide formal definitions, let the notation $(\mathtt{x} \ll \mathtt{y} \mid \mathtt{zs})$ mean "an occurrence of `x` appears physically before an occurrence of `y` in the collection `zs`." That is, $(\mathtt{x} \ll \mathtt{y} \mid \mathtt{zs})$ if and only if there are $i < j$ such that $\mathtt{x} = \mathtt{z}_i$ and $\mathtt{y} = \mathtt{z}_j$, where $\mathtt{z}_1, \mathtt{z}_2, ..., \mathtt{z}_n$ are the items in `zs` listed in their order of appearance in `zs`. Note that $(\mathtt{x} \ll \mathtt{x} \mid \mathtt{zs})$ if and only if `x` occurs at least twice in `zs`.

Also, a sorting key of a collection zs is a function $\phi(\cdot)$ with an associated linear ordering $<_\phi$ on its codomain such that, for every pair of items x and y in zs where $\phi(\mathtt{x}) \neq \phi(\mathtt{y})$, it is the case that $(\mathtt{x} \ll \mathtt{y} \mid \mathtt{zs})$ if and only if $\phi(\mathtt{x}) <_\phi \phi(\mathtt{y})$. Note that a collection may have zero, one, or more sorting keys. Two sorting keys $\phi(\cdot)$ and $\psi(\cdot)$ are said to have comparable codomains if their associated linear orderings are identical; i.e. for every z and z′, $\mathtt{z} <_\phi \mathtt{z}'$ if and only if $\mathtt{z} <_\psi \mathtt{z}'$. For convenience, in this situation, we write $<$ to refer to $<_\phi$ and $<_\psi$.

Definition 4.1 (Monotonicity of isBefore)**.** *An* isBefore *predicate is monotonic with respect to two collections* (xs, ys)*, which are not necessarily of the same type, if it satisfies the conditions below.*

1. *If* $(\mathtt{x} \ll \mathtt{x}' \mid \mathtt{xs})$*, then for all* y *in* ys*:* isBefore(y, x) *implies* isBefore(y, x′)*.*
2. *If* $(\mathtt{y}' \ll \mathtt{y} \mid \mathtt{ys})$*, then for all* x *in* xs*:* isBefore(y, x) *implies* isBefore(y′, x)*.*

Definition 4.2 (Antimonotonicity of canSee)**.** *Let* isBefore *be monotonic with respect to* (xs, ys)*. A* canSee *predicate is antimonotonic with respect to* isBefore *if it satisfies the conditions below.*

1. *If* $(\mathtt{x} \ll \mathtt{x}' \mid \mathtt{xs})$*, then for all* y *in* ys*:* isBefore(y, x) *and not* canSee(y, x) *implies not* canSee(y, x′)*.*
2. *If* $(\mathtt{y} \ll \mathtt{y}' \mid \mathtt{ys})$*, then for all* x *in* xs*: not* isBefore(y, x) *and not* canSee(y, x) *implies not* canSee(y′, x)*.*

To appreciate the monotonicity conditions, imagine two collections xs and ys are being merged without duplicate elimination into a combined list zs, in a manner that is consistent with the isBefore predicate and the physical order of appearance in xs and ys. To do this, let xs comprises $\mathtt{x}_1$, $\mathtt{x}_2$, ..., $\mathtt{x}_m$ as its elements and $(\mathtt{x}_1 \ll \mathtt{x}_2 \ll \cdots \ll \mathtt{x}_m|\mathtt{xs})$; let ys comprises $\mathtt{y}_1$, $\mathtt{y}_2$, ..., $\mathtt{y}_n$ as its elements and $(\mathtt{y}_1 \ll \mathtt{y}_2 \ll \cdots \ll \mathtt{y}_n|\mathtt{ys})$; and let $\mathtt{z}_i$ denote the ith element of zs. As there is no duplicate elimination, each $\mathtt{z}_i$ is necessarily a choice between some element $\mathtt{x}_j$ in xs and $\mathtt{y}_k$ in ys, and $i = j + k - 1$, unless all elements of xs or ys have already been chosen earlier. Let $\alpha(i)$ be the index of the element $\mathtt{x}_j$, i.e. j; and $\beta(i)$ be the index of the element $\mathtt{y}_k$, i.e. k. Obviously, $\alpha(1) = \beta(1) = 1$. And zs is necessarily constructed as follows: If $\alpha(i) > m$ or isBefore($\mathtt{y}_{\beta(i)}$, $\mathtt{x}_{\alpha(i)}$), then $\mathtt{z}_i = \mathtt{y}_{\beta(i)}$, $\alpha(i+1) = \alpha(i)$, and $\beta(i+1) = \beta(i) + 1$; otherwise, $\mathtt{z}_i = \mathtt{x}_{\alpha(i)}$, $\alpha(i+1) = \alpha(i) + 1$, and $\beta(i+1) = \beta(i)$.

Notice that in constructing zs above, only the isBefore predicate is used. The existence of a monotonic predicate isBefore with respect to (xs, ys) does not require xs and ys to be ordered by any sorting keys. For example, an "always true" isBefore predicate simply puts all elements of ys before all elements of xs when merging them into zs as described above. However, such trivial isBefore predicates have limited use.

When xs and ys are ordered by some sorting keys, more useful monotonic isBefore predicates are definable. For example, as an easy corollary of the construction of zs above, if xs and ys are ordered according to some sorting keys $\phi(\cdot)$ and $\psi(\cdot)$ with comparable codomains (i.e., $<_\phi$ and $<_\psi$ are identical and thus can be denoted simply as $<$), then a predicate defined as $\mathtt{isBefore(y, x)} = \psi(\mathtt{y}) < \phi(\mathtt{x})$ is guaranteed monotonic with respect to (xs, ys). To see this, without loss of generality, suppose for a contradiction that $(\mathtt{x}_i \ll \mathtt{x}_j|\mathtt{xs})$, $\phi(\mathtt{x}_i) \neq \phi(\mathtt{x}_j)$, and isBefore(y, $\mathtt{x}_i$), but not isBefore(y, $\mathtt{x}_j$). This means $\phi(\mathtt{x}_i) < \phi(\mathtt{x}_j)$,

$\psi(\mathtt{y}) < \phi(\mathtt{x}_i)$, but $\psi(\mathtt{y}) \not< \phi(\mathtt{x}_j)$. This gives the desired contradiction that $\phi(\mathtt{x}_j) < \phi(\mathtt{x}_j)$. This `isBefore(y, x)` $= \psi(\mathtt{y}) < \phi(\mathtt{x})$ is a natural bridge between the two sorted collections. Specifically, define $\omega(i) = \phi(\mathtt{z}_i)$ if $\mathtt{z}_i$ is from `xs` and $\omega(i) = \psi(\mathtt{z}_i)$ if $\mathtt{z}_i$ is from `ys`; and let $\omega(\mathtt{zs})$ denote the collection comprising $\omega(1)$, ..., $\omega(n+m)$ in this order. Then, $(\omega(i) \ll \omega(j) \mid \omega(\mathtt{zs}))$ implies $\omega(i) \leq \omega(j)$. That is, $\omega(\mathtt{zs})$ is linearly ordered by $<$, the associated linear ordering shared by the two sorting keys $\phi(\cdot)$ and $\psi(\cdot)$ of `xs` and `ys`.

Next, to appreciate the antimonotonicity conditions, one may eliminate the double negatives and read these antimonotonicity conditions as: (1) If `isBefore(y, x)` and $(\mathtt{x} \ll \mathtt{x}' \mid \mathtt{xs})$, then `canSee(y, x')` implies `canSee(y, x)`; and (2) If not `isBefore(y, x)` and $(\mathtt{y} \ll \mathtt{y}' \mid \mathtt{ys})$, then `canSee(y', x)` implies `canSee(y, x)`. Imagine that the `x`'s and `y`'s are placed on the same straight line, from left to right, in a manner consistent with `isBefore` (e.g., as explained above). Then, if `canSee` is antimonotonic to `isBefore`, its antimonotonicity implies a "right-sided" convexity. That is, if `y` can see an item `x` of `xs` to its right, then it can see all `xs` items between itself and this `x`. Similarly, if `x` can be seen by an item `y` of `ys` to its right, then it can be seen by all `ys` items between itself and this `y`. No "left-sided" convexity is required or implied however.

It follows that any `canSee` predicate which is reflexive and convex always satisfies the antimonotonicity conditions when `isBefore` satisfies the monotonicity conditions. So, we can try checking convexity and reflexivity of `canSee` first, which is a more intuitive task. Moreover, though this will not be discussed here, certain optimizations—which are useful in a parallel distributed setting—are enabled when `canSee` is reflexive and convex. Nonetheless, we must stress that the converse is not true. That is, an antimonotonic `canSee` predicate needs not be reflexive or convex; e.g., the `overlap(y, x)` predicate from Figure 1 is an example of a nonconvex antimonotonic predicate, and the inequality $m < n$ of two integers is an example of a nonreflexive convex antimonotonic predicate.

Proposition 4.3 (Reflexivity and convexity imply antimonotonicity)**.** *Let* `xs` *and* `ys` *be two collections, which are not necessarily of the same type. Let* `zs` *be a collection of some arbitrary type. Let* $\phi : \mathtt{xs} \to \mathtt{zs}$ *be a sorting key of* `xs` *and* $\psi : \mathtt{ys} \to \mathtt{zs}$ *be a sorting key of* `ys`*. Then* `isBefore` *is monotonic with respect to* $(\mathtt{xs}, \mathtt{ys})$*, and* `canSee` *is antimonotonic with respect to* `isBefore`*, if there are predicates* $<_{\mathtt{zs}}$ *and* $\lhd_{\mathtt{zs}}$ *such that all the conditions below are satisfied.*

1. ϕ *preserves order:* $(\mathtt{x} \ll \mathtt{x}' \mid \mathtt{xs})$ *implies* $(\phi(\mathtt{x}) \ll \phi(\mathtt{x}') \mid \mathtt{zs})$
2. ψ *preserves order:* $(\mathtt{y} \ll \mathtt{y}' \mid \mathtt{ys})$ *implies* $(\psi(\mathtt{y}) \ll \psi(\mathtt{y}') \mid \mathtt{zs})$
3. $<_{\mathtt{zs}}$ *preserves* `isBefore`*:* `isBefore(y, x)` *if and only if* $\psi(\mathtt{y}) <_{\mathtt{zs}} \phi(\mathtt{x})$
4. $<_{\mathtt{zs}}$ *is monotonic with respect to* $(\mathtt{zs}, \mathtt{zs})$
5. $\lhd_{\mathtt{zs}}$ *preserves* `canSee`*:* `canSee(y, x)` *if and only if* $\psi(\mathtt{y}) \lhd_{\mathtt{zs}} \phi(\mathtt{x})$
6. $\lhd_{\mathtt{zs}}$ *is reflexive: for all* `z` *in* `zs`*,* $\mathtt{z} \lhd_{\mathtt{zs}} \mathtt{z}'$
7. $\lhd_{\mathtt{zs}}$ *is convex: for all* $\mathtt{z}_0$ *in* `zs` *and* $(\mathtt{z} \ll \mathtt{z}' \ll \mathtt{z}'' \mid \mathtt{zs})$*,* $\mathtt{z} \lhd_{\mathtt{zs}} \mathtt{z}_0$ *and* $\mathtt{z}'' \lhd_{\mathtt{zs}} \mathtt{z}_0$ *implies* $\mathtt{z}' \lhd_{\mathtt{zs}} \mathtt{z}_0$*; and* $\mathtt{z}_0 \lhd_{\mathtt{zs}} \mathtt{z}$ *and* $\mathtt{z}_0 \lhd_{\mathtt{zs}} \mathtt{z}''$ *implies* $\mathtt{z}_0 \lhd_{\mathtt{zs}} \mathtt{z}'$

In particular, when $\mathtt{xs} = \mathtt{ys} = \mathtt{zs}$*, and* `isBefore` *is monotonic with respect to* $(\mathtt{xs}, \mathtt{ys})$ *and thus* $(\mathtt{zs}, \mathtt{zs})$*, conditions 1 to 5 above are trivially satisfied by setting the identity function as* ϕ *and* ψ*,* `isBefore` *as* $<_{\mathtt{zs}}$*, and* `canSee` *as* $\lhd_{\mathtt{zs}}$*. Thus, a reflexive and convex* `canSee` *is also antimonotonic.*

The antimonotonicity conditions provide us with two rules for moving on to the next `x` or the next `y`; cf. Figure 2. Specifically, according to Antimonotonicity Condition 1, when the current `y` in `ys` is before the current `x` in `xs`, and this `y` cannot "see" (i.e. does not match) this `x`, then this `y` cannot see any of the following items in `xs` either. Therefore, it is not necessary to try matching the current `y` to the rest of the items in `xs`, and we can move on to the next item in `ys`. On the other hand, according to Antimonotonicity Condition 2, when the current `y` in `ys` is not before the current `x` in `xs`, and this `y` cannot see this `x`, then all subsequent items in `ys` cannot see this `x` either. Therefore, it is not necessary to try matching the current `x` to the rest of the items in `ys`, and we can safely move on to the next item in `xs`.

When neither rule is triggered, regardless of whether the current `y` in `ys` is or is not before the current `x` in `xs`, this `y` can see this `x`. That is, we have a matching pair of `x` and `y` to perform some specified actions on. After the actions are performed, we can choose to move on to the next item in `xs` or in `ys`. In this work, we decide to keep the collection `xs` as the reference and to move on to the next item in the collection `ys`. Since the next item in `xs` may be an item that the current `y` can see, before moving on to the next item in `ys`, we should also "save" the current `y`; when we eventually move on to the next item in `xs`, we must remember to "rewind" our position in `ys` back to all these `y`'s saved during the processing of the current `x`.

Together, these conditions lead to what we call a *Synchrony fold*—the `syncFold` function defined in Figure 3—which iterates on two collections in synchrony.

What does `syncFold(f, e, bf, cs)(xs, ys)` do? To answer this question, consider the function `slowFold(f, e, cs)(xs, ys)` which is also defined in Figure 3. The function `slowFold(f, e, cs)(xs, ys)` first initializes an internal variable `acc` to `e`; then iterates through every pair of `x` in `xs` and `y` in `ys`, and updates `acc` to `f(x, y, acc)` whenever `cs(y,x)`; at the end of the iteration, it outputs the value of `acc`.

Remarkably, when `bf` is monotonic with respect to $(\texttt{xs}, \texttt{ys})$ and `cs` is antimonotonic with respect to `bf`, `syncFold(f, e, bf, cs)(xs, ys)` computes the same result as `slowFold(f, e, cs)(xs, ys)`. Furthermore, `syncFold` has a potentially linear complexity $O(|\texttt{xs}| + k|\texttt{ys}|)$ in terms of number of calls to the function f, when `cs` has degree $< k$ in the sense that $|\{\texttt{x} \in \texttt{xs} \text{ such that } \texttt{cs(y, x)}\}| < k$ for each `y` in `ys`. Whereas, `slowFold` has quadratic complexity $O(|\texttt{xs}| \cdot |\texttt{ys}|)$.

Theorem 4.4 (Synchrony fold)**.** *Suppose* `isBefore` *is monotonic with respect to* $(\texttt{xs}, \texttt{ys})$ *and* `canSee` *is antimonotonic with respect to* `isBefore`*.*

1. `syncFold(f, e, isBefore, canSee)(xs, ys)` $=$ `slowFold(f, e, canSee)(xs, ys)`.
2. `slowFold(f, e, canSee)(xs, ys)` *calls the function* `f` *a total of* $|\texttt{xs}| \cdot |\texttt{ys}|$ *number of times.*
3. `syncFold(f, e, isBefore, canSee)(xs, ys)` *calls the function* `f` *at most* $|\texttt{xs}| + k|\texttt{y}|$ *number of times, if* `canSee` *has degree* $< k$ *with respect to* $(\texttt{xs}, \texttt{ys})$.

Proof For Part 1, consider the function `aux(xs, ys, zs, acc)` in `syncFold`. Suppose `isBefore` is monotonic with respect to $(\texttt{xs}, \texttt{zs ++: ys})$, and `canSee` is antimonotonic with

```
def syncFold[A,B,C]
  (f: (A,B,C) => C, e: C, bf: (B,A) => Boolean, cs: (B,A) => Boolean)
  (xs: Vec[A], ys: Vec[B])
: C = {
  // Requires: bf monotonic wrt (xs,ys); cs antimonotonic wrt bf.
  // Assumes:  isEmpty, head, tail are constant time;
  //           prepend (++:) is linear in its left argument;
  //           single-item postpend (:+) is constant time.

  def aux(xs: Vec[A], ys: Vec[B], zs: Vec[B], acc: C): C =
    if (xs.isEmpty) acc
    else if (ys.isEmpty && zs.isEmpty) acc
    else if (ys.isEmpty) aux(xs.tail, zs, Vec(), acc)
    else {
      val (x, y) = (xs.head, ys.head)
      (bf(y, x), cs(y, x)) match {
        case (true, false) =>
          // Antimonotonicity Condition 1:
          // bf(y,x) & !cs(y,x) => all x' after x: !cs(y,x')
          // So, y can be discarded safely; move on to next y.
          aux(xs, ys.tail, zs, acc)

        case (false, false) =>
          // Antimonotonicity Condition 2:
          // !bf(y,x) & !cs(y,x) => all y' after y: !cs(y',x)
          // So x can be discarded safely. But the next x may
          // still be able to see some y saved earlier in zs.
          aux(xs.tail, zs ++: ys, Vec(), acc)

        case (_, true) =>
          // At this point, cs(y,x); so process (x,y) using f.
          // Save this y as it may see next x; move on to next y.
          aux(xs, ys.tail, zs :+ y, f(x, y, acc))
      }
    }

  aux(xs, ys, Vec(), e)
}


def slowFold[A,B,C]
  (f: (A,B,C) => C, e: C, cs: (B,A) => Boolean)
  (xs: Vec[A], ys: Vec[B])
: C = {
  var acc: C = e
  for (x <- xs; y <- ys; if cs(y, x)) { acc = f(x, y, acc) }
  return acc
}
```

Fig. 3. Definitions of `syncFold` and `slowFold`. These two programs compute the same results when `bf` is monotonic with respect to (`xs`, `ys`) and `cs` is antimonotonic with respect to `bf`. However, `syncFold` is more efficient than `slowFold`.

respect to `isBefore`. If `xs` is non-empty, let `x` be `xs.head`, and z_1, ..., z_n be the items in `zs` such that `canSee(`z_1`, x)`, ..., `canSee(`z_n`, x)`, and `acc` = `f(x,` z_n`, ... f(x,` z_1`, e) ... )`. If `xs` is empty, let `acc` = `e`. Then, an induction on (|`xs`|, |`ys`|) shows that

```
  aux(xs, ys, zs, acc)
= slowFold(f, e, canSee)(xs, zs ++: ys)
```

So,

```
  syncFold(f, e, isBefore, canSee)(xs, ys)
= aux(xs, ys, Vec(), e)
= slowFold(f, e, canSee)(xs, ys)
```

For Part 2, it is obvious that `slowFold(f, e, canSee)(xs, ys)` calls the function f a total of $|\texttt{xs}| \cdot |\texttt{ys}|$ number of times.

For Part 3, on each call to `aux` in `syncFold`, either `xs` or `ys` is shortened by 1 item. This gives $|\texttt{xs}| + |\texttt{ys}|$ calls to `aux`. In some calls, `ys` is prepended with `zs`. Recall the assumption that `canSee` has degree $< k$. Thus, each item in `ys` can see fewer than k items in `xs`. So, the total size of `zs` summed over all the calls to `aux` is at most $(k-1)|\texttt{ys}|$; these are the maximum number of additional calls to `aux`. Therefore, the total number of calls to `aux`, and thus to `f`, is at most $|\texttt{xs}| + k|\texttt{ys}|$. ■

4.2 Second Synchrony fold

`SyncFold(f, e, bf, cs)(xs, ys)` discards items in `xs` that no item in `ys` sees. This may not be desired in some situations, e.g., when someone actually wants to retrieve those items in `xs` that no item in `ys` sees. Also, `syncFold` pairs up each `x` in `xs` with each `y` in `ys` that sees it, and applies the function `f` on these pairs one by one. This may not be convenient in some situations; e.g., when someone wants to count the number of `y`'s that see an `x`. Hence, it might be useful to also provide a second Synchrony fold function `syncFoldGrp` which processes, as a group, those `y`'s that see an `x`.

An astute reader might have realised that, in the definition provided in Figure 3, `syncFold` keeps the `y`'s that can see the current `x` in the collection `zs`. So, as defined in Figure 4, `syncFoldGrp(f, e, bf, cs)(xs, ys)` is just `syncFold` with `f` applied to `(x, zs, acc)` instead of `(x, y, acc)`. The function `syncFoldGrp(f, e, bf, cs)(xs, ys)` computes the same result as `slowFoldGrp(f, e, cs)(xs, ys)`, which is also defined in Figure 4 and is much easier to understand. However, while the former can be linear in time complexity, the latter is quadratic.

Theorem 4.5 (Second Synchrony fold)**.** *Suppose* `isBefore` *is monotonic with respect to* $(\texttt{xs}, \texttt{ys})$ *and* `canSee` *is antimonotonic with respect to* `isBefore`*. Then,*

1. `syncFoldGrp(f, e, isBefore, canSee)(xs, ys)` = `slowFoldGrp(f, e, canSee)(xs, ys)`.

Suppose further that `canSee` *has degree* $< k$ *with respect to* $(\texttt{xs}, \texttt{ys})$*, and* `f` *has linear time complexity in its second argument, and other arguments have neglible influence on* `f`*'s time complexity. Then,*

2. `slowFoldGrp(f, e, canSee)(xs, ys)` *has time complexity* $O((|\texttt{xs}| + k)|\texttt{ys}|)$.
3. `syncFoldGrp(f, e, isBefore, canSee)(xs, ys)` *has time complexity* $O(|\texttt{xs}| + 2k|\texttt{ys}|)$.

Proof For Part 1, consider the function `aux(xs, ys, zs, acc)` in `syncFoldGrp`. Suppose `isBefore` is monotonic with respect to $(\texttt{xs}, \texttt{zs ++: ys})$, and `canSee` is antimonotonic with

```
def syncFoldGrp[A,B,C]
  (f: (A,Vec[B],C)=>C, e: C, bf: (B,A)=>Boolean, cs: (B,A)=>Boolean)
  (xs: Vec[A], ys: Vec[B])
: C = {
  // Requires: bf monotonic wrt (xs,ys) & cs antimonotonic wrt bf.

  def aux(xs: Vec[A], ys: Vec[B], zs: Vec[B], acc: C): C =
    if (xs.isEmpty) acc
    else if (ys.isEmpty && zs.isEmpty) acc
    else if (ys.isEmpty) aux(xs.tail, zs, Vec(), f(xs.head, zs, acc))
    else {
      val (x,y) = (xs.head, ys.head)
      (bf(y, x), cs(y, x)) match {
        case (true, false) =>
          // Antimonotonicity Condition 1:
          // bf(y,x) & !cs(y,x) => all x' after x: !cs(y,x')
          // So, y can be discarded safely; move on to next y.
          aux(xs, ys.tail, zs, acc)

        case (false, false) =>
          // Antimonotonicity Condition 2:
          // !bf(y,x) & !cs(y,x) => all y' after y: !cs(y',x)
          // So, x can be discarded. And the y accumulated in zs
          // should now be processed by f in one go. Note: the
          // next x may be able to see some y accumulated in zs.
          aux(xs.tail, zs ++: ys, Vec(), f(x, zs, acc))

        case (_, true) =>
          // At this point, cs(y,x).
          // Accumulate this y in zs; move on to next y.
          aux(xs, ys.tail, zs :+ y, acc)
      }
    }

  aux(xs, ys, Vec(), e)
}


def slowFoldGrp[A,B,C]
  (f: (A,Vec[B],C) => C, e: C, cs: (B,A) => Boolean)
  (xs: Vec[A], ys: Vec[B])
: C = {
  var acc: C = e
  for (x <- xs; zs = for (y <- ys; if cs(y, x)) yield y) {
    acc = f(x, zs, acc)
  }
  return acc
}
```

Fig. 4. Definitions of `syncFoldGrp` and `slowFoldGrp`. They compute the same results when `bf` is monotonic with respect to (`xs`, `ys`) and `cs` is antimonotonic with respect to `bf`. However, `syncFoldGrp` is more efficient than `slowFoldGrp`.

respect to `isBefore`. Suppose also that `canSee(z, x)` for each `z` in `zs`, when `xs` is non-empty and `x` is `xs.head`. Then, an induction on $(|\texttt{xs}|, |\texttt{ys}|)$ shows that

```
  aux(xs, ys, zs, acc)
= slowFoldGrp(f, acc, canSee)(xs, zs ++: ys)
```

So,

```
  syncFoldGrp(f, e, isBefore, canSee)(xs, ys)
= aux(xs, ys, Vec(), e)
= slowFoldGrp(f, e, canSee)(xs, ys)
```

For Part 2, the theorem assumes that `canSee` has degree $< k$, and `f` has time complexity linear in its second argument and independent of its other arguments. The first assumption implies that the total size of `zs` over all the calls to `f` is at most $k|\texttt{ys}|$. The second assumption implies that the total time complexity due to calls to `f` is $O(k|\texttt{ys}|)$. The nested loops of `slowFoldGrp`, excluding calls to `f`, has $O(|\texttt{xs}| \cdot |\texttt{ys}|)$ time complexity. Thus, summing these two components gives a quadratic time complexity, $O((|\texttt{xs}| + k)|\texttt{ys}|)$.

For Part 3, again recall the two assumptions of the theorem, viz. `canSee` has degree $< k$, and `f` has time complexity linear in its second argument and independent of its other arguments. The first assumption implies that the total size of `zs` over all the calls to `f` is at most $k|\texttt{ys}|$. The second assumption implies that the total time complexity due to calls to `f` is $O(k|\texttt{ys}|)$. In addition, as in `syncFold`, there are at most $|\texttt{xs}| + k|\texttt{ys}|$ calls to `aux` in `syncFoldGrp`. Summing these gives a linear time complexity, $O(|\texttt{xs}| + 2k|\texttt{ys}|)$. ■

Now, let `snoc(x, zs, a) = a :+ (x, zs)` add `(x, zs)` to the end of a collection `a`. Then,

```
  slowFoldGrp(snoc, Vec(), cs)(Vec(x), ys)
= Vec((x, for (y <- ys; if cs(y, x)) yield y))

  slowFoldGrp(snoc, Vec(), cs)(xs, ys)
= for (x <- xs; (x′, zs) <- slowFoldGrp(snoc, Vec(), cs)(Vec(x), ys)) yield (x′, zs)
```

The corollary below now follows from Theorem 4.5. This corollary is helpful for a deeper understanding of `syncFoldGrp`, leading later to the design of Synchrony iterator in Section 6.

Corollary 4.6. *Let* `isBefore` *be monotonic with respect to* $(\texttt{xs}, \texttt{ys})$*, and* `canSee` *be antimonotonic with respect to* `isBefore`*. Let* `snoc(x, zs, a) = a :+ (x, zs)`*. Then,*

```
  syncFoldGrp(snoc, Vec(), isBefore, canSee)(xs, ys)
= for ( x <- xs; (x′, zs) <- syncFoldGrp(snoc, Vec(), isBefore, canSee)(Vec(x), ys)
  yield (x′, zs)
```

4.3 *Synchrony fold vs* `foldLeft`

As mentioned earlier, `syncFold` and `syncFoldGrp` are generalizations of `foldLeft`. In particular, as shown below, `foldLeft` is definable via either of them.

```
  xs.foldLeft(e)(g)
= syncFold((x,_,a)=>g(a,x), e, (_,_)=>true, (_,_)=>true)(xs, Vec(()))
= syncFoldGrp((x,_,a)=>g(a,x), e, (_,_)=>true, (_,_)=>true)(xs,Vec(()))
```

Furthermore, both definitions are as efficient as the implementation of `foldLeft` in collection-type libraries; e.g., if the function `g` above has $O(1)$ time complexity, then both implementations of `foldLeft` above have $O(|\texttt{xs}|)$ time complexity, same as any typical implementation of `foldLeft` in collection-type libraries of modern programming languages.

At the same time, functions expressible by `syncFold` and `syncFoldGrp` are also expressible in first-order restricted Scala when `foldLeft` is available. Let `isBefore` be monotonic with respect to (`xs`, `ys`), and `canSee` be antimonotonic with respect to `isBefore`. Then,

```
  syncFold(f, e, isBefore, canSee)(xs, ys)
= xs.foldLeft(e)((a, x) =>
      ys.foldLeft(a)((a′, y) => if (canSee(y, x)) f(x, y, a′) else a′))

  syncFoldGrp(f′, e, isBefore, canSee)(xs, ys)
= xs.foldLeft(e)((a, x) =>
      f′(x, for (y <- ys; if canSee(y, x)) yield y, a))
```

These implementations of `syncFold` and `syncFoldGrp` in terms of `foldLeft` are quadratic in time complexity. They are also somewhat more convoluted than the implementations of `foldLeft` in terms of `syncFold` and `syncFoldGrp`. Perhaps more ingenious programmers can find some simpler ways of implementing `syncFold` and `syncFoldGrp` solely in terms of `foldLeft`. Unfortunately, due to Lemma 3.3, there is no way they can find an efficient linear-time implementation of either one using `foldLeft` alone under the first-order restriction.

Proposition 4.7 (`SyncFold` and `syncFoldGrp` are conservative extensions of `foldLeft`)**.** *The extensional expressive power of Scala under the first-order restriction, when* `foldLeft` *is available, is the same with or without* `syncFold` *and* `syncFoldGrp`*. However, more efficient algorithms for some functions (e.g., a linear-time algorithm for low-selectivity join) can be defined using* `syncFold` *and* `syncFoldGrp` *than using* `foldLeft` *in Scala under the first-order restriction.*

It is worth noting that `syncFold(f, e, isBefore, canSee)(xs, ys)` and the expression `syncFoldGrp((x, zs, a) => zs.foldLeft(a)((a′, z) => f(z, a′)), e, isBefore, canSee)(xs, ys)` compute the same function at comparable time complexity. So, `syncFold` can be defined efficiently using `syncFoldGrp`. Similarly, `SyncFoldGrp` can also be implemented at comparable time complexity using `syncFold`. To wit, as presented later in Section 5.1, efficient `takeWhile` and `dropWhile` are definable by `syncFold`; in turn, efficient `syncFoldGrp` needs only a straightforward modification to the implementation—using `foldLeft`, `takeWhile`, and `dropWhile`—of the function `groups2` shown in Figure 13 of Section 8.1.

4.4 Synchrony fold in action

Linear time complexity for the example from Section 2, `ov1(xs, ys)`, can be achieved using `syncFold`. The codes for `ov3(xs, ys)` below shows that a user-programmer only has to provide straightforward definitions for the `isBefore` and `canSee` predicates; for this example, these are the `isBefore` and `overlap` predicates defined earlier in Figure 1. There is no worry about getting the "synchronized" iteration of `xs` and `ys` right, as `syncFold` takes care of this already. The linear time complexity is easily appreciated using Theorem 4.4 when `overlap` has a low degree with respect to (`xs`, `ys`), i.e. each event in `ys` overlaps few events in `xs`.

```
def ov3(xs: Vec[Event], ys: Vec[Event]) = {
  // Requires: xs and ys are sorted lexicographically by (start, end).
  // Note: isBefore and overlap are as defined in Figure 1.
```

```
  def f(x: Event, y: Event, acc: Vec[(Event, Event)]) = acc :+ (x, y)
  syncFold(f, Vec(), isBefore, overlap)(xs, ys)
}
```

There is a loose end to be tied up in the example above, viz. verifying that `isBefore` is monotonic with respect to `(xs, ys)` and `overlap` is antimonotonic with respect to `isBefore`. This is omitted here, as it is straightforward under the assumption that `(xs, ys)` are lexicographically ordered by the `start` and `end` point of their events.

As an example of `syncFoldGrp`, it is used below to count in potentially linear time the number of events in `ys` that each event in `xs` overlaps with. The linear time complexity follows from Theorem 4.5.

```
def ovCount(xs: Vec[Event], ys: Vec[Event]): Vec[(Event,Int)] = {
  // Requires: xs and ys are sorted lexicographically by (start, end).
  // Note: isBefore and overlap are as defined in Figure 1.
  def f(x: Event, zs: Vec[Event], acc: Vec[(Event,Int)]) = {
    acc :+ (x, zs.length)
  }
  syncFoldGrp(f, Vec(), isBefore, overlap)(xs, ys)
}
```

Comparing `ov3(xs, ys)` above and `ov2(xs, ys)` from Figure 1, the design of Synchrony fold makes clear three orthogonal aspects of the event-overlap example: connecting the orderings on the two collections, identifying matching pairs, and acting on matching pairs. With regard to connecting the orderings on the two collections, the "navigation" is captured by the `isBefore` predicate. With regard to identification of matching pairs, it is captured by the `canSee` predicate (i.e. `overlap`). Finally, with regard to action on matching pairs, this is captured by the function `f`. Making these three orthogonal aspects explicit brings about a more concise and precise understanding (Schmidt, 1986; Hunt & Thomas, 2000; Sebesta, 2010). For example, assuming `isBefore` is monotonic with respect to `(xs, ys)` and `canSee` is antimonotonic with respect to `isBefore`, one can read `syncFold(f, e, isBefore, canSee)(xs, ys)` simply as "for each pair in `(xs, ys)` satisfying `canSee`, do `f` on it." Hopefully, this clarity makes it easier to see mistakes, and thus easier to write programs correctly.

This simple way to read Synchrony fold programs was in fact formalized earlier via Theorem 4.4 and 4.5. These two theorems reveal the extensional equivalence of `syncFold` and `slowFold`, and of `syncFoldGrp` and `slowFoldGrp`. While `slowFold` and `slowFoldGrp` are intuitive, they use some local side effects. Now, comparing `ov3(xs, ys)` and `ov1(xs, ys)`, a straightforward relationship between a restricted form of `syncFold` and `syncFoldGrp` and comprehension syntax can be further discerned below; this time without side effects. This relationship also shows that any join whose predicate `p(y, x)` can be decomposed into an antimonotonic predicate `canSee(y, x)` and a residual predicate `h(y, x)`, can be implemented using `syncFold` and `syncFoldGrp` efficiently.

Proposition 4.8 (Comprehending `syncFold` and `syncFoldGrp`)**.** *Suppose* `xs` *and* `ys` *are two collections,* `isBefore` *is monotonic with respect to* `(xs, ys)`*, and* `canSee` *is antimonotonic with respect to* `isBefore`*. Then, these three Scala programs express the same function:*

1. `for (x <- xs; y <- ys; if canSee(y, x) && h(y, x)) yield g(x, y)`

2. `syncFold(f, Vec(), isBefore, canSee)(xs, ys)`, *where*
   `f(x, y, acc)` = `if (h(y,x)) { acc :+ g(x, y) } else acc`
3. `syncFoldGrp(f′, Vec(), isBefore, canSee)(xs, ys)`, *where*
   `f′(x, zs, acc)` = `acc :++ for (z <- zs; if h(z, x)) yield g(x, z)`

However, when `canSee` *has a low degree and* `g` *and* `h` *have* $O(1)$ *time complexity, the first program is quadratic while the second and third programs are linear in their time complexity with respect to* $|$`xs`$|$ *and* $|$`ys`$|$.

5 Synchrony generator

Lemma 3.1 indicates that an intensional expressiveness gap already exists in first-order restricted Scala sans library functions. And Lemma 3.3 further indicates that this same gap exists practically unmitigated when first-order restricted Scala is augmented with `foldLeft`. On the one hand, Proposition 4.7 shows that the two Synchrony folds are conservative extensions of first-order restricted Scala augmented with `foldLeft`, and significantly increases the algorithmic richness of this fragment of Scala. On the other hand, Proposition 4.7 also means that Synchrony fold is an overkill as a solution for this gap which originated at the level of first-order restricted Scala without library functions, since Synchrony fold adds much extra extensional expressive power to this fragment of Scala while fixing its intensional expressive power gap.

This section identifies a restriction on Synchrony fold to fix this gap at its root, i.e. at the level of unaugmented first-order restricted Scala. The significance of this restricted form, viz. *Synchrony generator*, in the context of database joins is also discussed.

5.1 Deriving Synchrony generator

As mentioned, we wish to identify some restriction on Synchrony fold to cut its extensional expressive power to that of first-order restricted Scala sans library functions. Proposition 4.8 suggests the two solutions `syncMap` and `syncFlatMap`, shown in Figure 5.

Ignoring efficiency issues, the functions expressible by `syncMap` and `syncFlatMap` are already expressible just using comprehension syntax, when `isBefore` is monotonic with respect to (`xs`, `ys`) and `canSee` is antimonotonic with respect to `isBefore`. Specifically,

```
  syncMap(f, isBefore, canSee)(xs, ys)
= for (x <- xs; y <- ys; if canSee(y, x)) yield f(x, y)

  syncFlatMap(f, isBefore, canSee)(xs, ys)
= for (x <- xs; z <- f(x, for (y <- ys; if canSee(y, x)) yield y))
  yield z
```

Thus, `syncMap` and `syncFlatMap` do not add extensional expressive power to first-order restricted Scala sans library functions, but add to it sufficient algorithmic power to implement efficient low-selectivity joins.

In fact, an even more stringent restriction, the *Synchrony generators*, `syncGen` and `syncGenGrp`, also depicted in Figure 5, can provide the same extra intensional expressive power as `syncMap` and `syncFlatMap`. This is because

```
def syncMap[A,B,C]
  (f: (A,B)=>C, bf: (B,A)=>Boolean, cs: (B,A)=>Boolean)
  (xs: Vec[A], ys: Vec[B])
: Vec[C] = {
  // Requires: bf monotonic wrt (xs, ys); cs antimonotonic wrt bf.
  val step = (x: A, y: B, acc: Vec[C]) => acc :+ f(x,y)
  syncFold(step, Vec(), bf, cs)(xs, ys)
}

def syncFlatMap[A,B,C]
  (f: (A,Vec[B])=>Vec[C], bf: (B,A)=>Boolean, cs: (B,A)=>Boolean)
  (xs: Vec[A], ys: Vec[B])
: Vec[C] = {
  // Requires: bf monotonic wrt (xs, ys); cs antimonotonic wrt bf.
  val step = (x: A, zs: Vec[B], acc: Vec[C]) => acc :++ f(x,zs)
  syncFoldGrp(step, Vec(), bf, cs)(xs, ys)
}

def syncGen[A,B]
  (isBefore: (B,A) => Boolean, canSee: (B,A) => Boolean)
  (xs: Vec[A], ys: Vec[B])
: Vec[(A,B)] = {
  // Requires: bf monotonic wrt (xs, ys); cs antimonotonic wrt bf.
  val step = (x: A, y: B, acc: Vec[(A,B)]) => acc :+ (x, y)
  val e: Vec[(A,B)] = Vec()
  syncFold(step, e, isBefore, canSee)(xs, ys)
}

def syncGenGrp[A,B]
  (bf: (B,A) => Boolean, cs: (B,A) => Boolean)
  (xs: Vec[A], ys: Vec[B])
: Vec[(A,Vec[B])] = {
  // Requires: bf monotonic wrt (xs, ys); cs antimonotonic wrt bf.
  val step = (x: A, zs: Vec[B], acc: Vec[(A,Vec[B])]) => acc :+ (x,zs)
  val e: Vec[(A,Vec[B])] = Vec()
  syncFoldGrp(step, e, bf, cs)(xs, ys)
}
```

Fig. 5. Definitions of `syncMap`, `syncFlatMap`, `syncGen` and `SyncGenGrp`.

```
  syncFlatMap(f, isBefore, canSee)(xs, ys)
= for ((x, zs) <- syncGenGrp(isBefore, canSee)(xs, ys); z <- f(x, zs)) yield z

  syncMap(f, isBefore, canSee)(xs, ys)
= for ((x, y) <- syncGen(isBefore, canSee)(xs, ys)) yield f(x, y)
```

Strictly speaking, `syncGenGrp` is not first-order restricted as it returns a nested collection. However, let us constrain it to be used strictly as a generator `(x, zs) <- syncGenGrp(bf, cs)(xs, ys)` in a comprehension construct, with the understanding that `for ((x, zs) <- syncGenGrp(bf, cs)(xs, ys); ... ) yield e` is desugared to `syncFlatMap((x, zs) => for ( ... ) yield e, bf, cs)(xs, ys)`. With this constraint, `syncGenGrp` can justifiably be viewed as a first-order construct, as it becomes mere syntactic sugar which gets desugared into a first-order construct.

As shown earlier, `syncMap` and `syncFlatMap` are expressible as functions in comprehension syntax. And `syncGen` and `syncGenGrp` are desugared into `syncMap` and `syncFlatMap`. So, the theorem below follows.

Theorem 5.1. *The extensional expressive power of Scala under the first-order restriction, is the same with or without any of* `syncMap`, `syncFlatMap`, `syncGen`, *and* `syncGenGrp`. *However, more efficient algorithms for some functions (e.g., a linear-time algorithm for low-selectivity join) can be defined when any of* `syncMap`, `syncFlatMap`, `syncGen`, *and* `syncGenGrp` *is made available in this fragment of Scala.*

For illustration, the function `ov1(xs, ys)` from Figure 1 is expressed below using `syncGen`. This version, `ov4(xs, ys)`, as with `ov3(xs, ys)` in Section 4.4, has linear time complexity when selectivity is low.

```
def ov4(xs: Vec[Event], ys: Vec[Event]): Vec[(Event, Event)] = {
  // Requires: xs and ys sorted lexicographically by (start, end).
  // Note: isBefore and overlap are as defined in Figure 1.
  syncGen(isBefore, overlap)(xs, ys)
}
```

Recall also Lemma 3.2 that $\mathscr{NRC}_1(\leq, \texttt{takeWhile}, \texttt{dropWhile}, \texttt{sort})$ cannot realise efficient low-selectivity joins. Therefore, first-order restricted Scala augmented with `takeWhile` and `dropWhile`, by themselves, cannot implement `syncGen` and `syncGenGrp` efficiently. On the other hand, both `takeWhile` and `dropWhile` can be realised quite efficiently and succinctly using either Synchrony generator. For example,

```
  for (x <- xs.takeWhile(p)) yield f(x)
= for ((_, x) <- syncGen((_,_)=>false, (x,_)=>p(x))(Vec(()), xs)) yield f(x)

  for (x <- xs.dropWhile(p)) yield f(x)
= for ((x,_) <- syncGen((y,x)=>!y && !p(x), (y,_)=>y)(xs, Vec(false,true))) yield f(x)
```

5.2 Synchrony generator vs database merge join

As mentioned in Section 2, a database join having a join predicate comprising entirely of equality tests is an equijoin, and those comprising entirely of inequality tests is a non-equijoin. A relational database system executes joins using a variety of strategies (Blasgen & Eswaran, 1977; Mishra & Eich, 1992; Silberschatz *et al.*, 2016). Where possible, a relational database system decomposes a join predicate into an equijoin part and a residual part; it then executes the equijoin part using either an index join (if suitable indices are available), or a merge join (if indices are not available but the relations are already appropriately sorted), or a sort-merge join or a hash join (if indices are not available and the relations are not already sorted); finally, it executes the residual part as a selection predicate on the result of the equijoin. So, the time complexity is always linear or at worst linearithmic for a join which has an equijoin part that has low selectivity. For a non-equijoin, most relational database systems execute it using nested loops, which have quadratic time complexity. However, some relational database systems can execute some restricted forms of non-equijoin, such as a band join `x.a` $\leq$ `y.b` $\leq$ `x.c`, more efficiently (e.g., in linear time, when the band join predicate `x.a` $\leq$ `y.b` $\leq$ `x.c` has low selectivity.)

The Synchrony generator `syncGen(isBefore, canSee)` is closely related to, and is an elegant generalization of, the merge join used in relational database systems. In relational database systems, the merge join is always applied on a pair of relational tables `xs` and `ys`

which are sorted according to some sorting keys $\phi(\cdot)$ and $\psi(\cdot)$ with comparable codomains. This induces a linear ordering `isBefore(y, x)` $= \psi(\mathtt{y}) < \phi(\mathtt{x})$ on items in the two tables. So, by construction, this `isBefore` predicate is monotonic with respect to (`xs`, `ys`).

For the standard merge join (Silberschatz *et al.*, 2016), the join predicate `canSee` must comprise entirely of equality tests (i.e. an equijoin). So, `canSee` is reflexive and convex; thus, by Proposition 4.3, it is antimonotonic with respect to `isBefore`. All relational database systems also support a variant of the merge join where the join predicate is a single inequality like `canSee(y, x)` $=$ `x.a` $<$ `y.b` or `canSee(y, x)` $=$ `x.a` $\leq$ `y.b`. The former is antimonotonic and convex but not reflexive, the latter is convex and reflexive and thus also antimonotonic. Some database systems support a range join predicate of the form `canSee(y, x)` $=$ `x.a - `$\varepsilon \leq$ `y.b` $\leq$ `x.a + `ε; cf. DeWitt *et al.* (1991). This is a reflexive and convex predicate. Thus, by Proposition 4.3, it is antimonotonic with respect to `isBefore`. Newer database systems support a band join predicate of the form `canSee(y, x)` $=$ `x.a` $\leq$ `y.b` $\leq$ `x.c`. This predicate is antimonotonic but not convex. Some database systems support an interval join predicate of the form `canSee(y, x)` $=$ `x.a` $\leq$ `y.b && y.c` $\leq$ `x.d` on some special data types, such as those associated with time periods, where the constraints `x.a` $\leq$ `x.d` and `y.c` $\leq$ `y.b` are known or enforced by these database systems; cf. Piatov *et al.* (2016) and Dignoes *et al.* (2021). This predicate, taking into account the two associated constraints, is antimonotonic but not convex.

As all these `canSee` predicates are antimonotonic, they can be used legitimately in `syncGen(isBefore, canSee)(xs, ys)`. And this computes the corresponding equijoin, single-inequality merge join, range join, band join, and interval join. Clearly, the monotonicity of the `isBefore` predicate and the antimonotonicity of the `canSee` predicate constitute a more general and more elegant condition for correctness than the adhoc syntactic forms required by current formulations of equijoin, single-inequality merge join, range join, band join, and interval join.

There have been many works introducing join algorithms in the database community to handle non-equijoin, from early studies by DeWitt *et al.* (1991) to recent studies by Piatov *et al.* (2016) and Dignoes *et al.* (2021). These works generally require a combination of new data structures, new evaluation techniques, and even exploitation of hardware features of modern CPU architectures. These are tools which are not part of the repertoire of an average programmer. Moreover, these works consider only some syntactic forms. In contrast, Synchrony generator efficiently and uniformly implements a more general class of non-equijoin without requiring any of these. This makes Synchrony generator rather appealing as an addition to collection-type function libraries of programming languages.

Moreover, by Theorem 4.4, the time complexity of `syncGen(isBefore, canSee)(xs, ys)` is $O(|\mathtt{xs}| + k|\mathtt{ys}|)$ where k is the degree of the `canSee` predicate. It is worth noting that the size of the result of `syncGen(isBefore, canSee)(xs, ys)` is also $O(|\mathtt{xs}| + k|\mathtt{ys}|)$, which obviously constitutes a lowerbound on the efficiency of any algorithm for computing the same join. So, despite Synchrony generator being much simpler and more general than earlier algorithms for various more restricted forms of non-equijoin, the time complexity of `syncGen(isBefore, canSee)(xs, ys)` is already asymptotically optimal. Even more impressive, it does this while staying strictly within the extensional expressive power of first-order restricted Scala unaugmented with any library function.

```
case class Event(start: Int, end: Int, id: String)
// Constraint: start < end

val isBeforeWithId = (y: Event, x: Event) => {
  (y.id < x.id) ||
  (y.id == x.id && y.start < x.start) ||
  (y.id == x.id && y.start == x.start && y.end < x.end)
}

val overlapWithId(y: Event, x: Event) => {
  (y.id == x.id) &&
  (x.start < y.end && y.start < x.end)
}

def ovWithId(xs: Vec[Event], ys: Vec[Event]) = {
  // Requires: xs and ys are sorted by (id, start, end)
  syncGen(isBeforeWithId, overlapWithId)(xs, ys)
}

// Query1: ovWithId directly translated into SQL
SELECT x.*, y.*
FROM xs AS x, ys AS y
WHERE y.id = x.id AND x.start < y.end AND y.start < x.end

// Query2: ovWithId implemented as "Union of band joins" in SQL
SELECT x.*, y.*
FROM xs AS x JOIN ys AS y ON y.start < x.start AND x.start < y.end
WHERE y.id = x.id
UNION ALL
SELECT x.*, y.*
FROM xs AS x JOIN ys AS y ON x.start <= y.start AND y.start < x.end
WHERE y.id = x.id
```

Fig. 6. A variation of the event-overlap example. `ovWithId(xs, ys)` computes the same function as the two SQL queries on inputs `xs` and `ys` which are sorted lexicographically by `(id, start, end)`.

Therefore, the formulation of Synchrony generator and the monotonicity and anti-monotonicity conditions on the associated `isBefore` and `canSee` predicates add conceptual elegance and algorithmic clarity in characterizing and generalizing the merge join.

For a further appreciation of what this brings, consider a slight variation of the event-overlap example. As shown in Figure 6, this time, events are categorized by their `id` attribute (where there can be many events of each category), and are ordered lexicographically by their `id`, `start`, and `end` attributes. An event `y` is now considered before another event `x` either when `y` has a smaller category `id` than `x`, or they have the same category `id` and `y` starts before `x`, or they have the same category `id` and start together but `y` ends earlier than `x`, as defined by the function `isBeforeWithId` in the figure. Similarly, two events now are considered overlapping only when they have the same category `id` and they overlap in time, as defined by the function `overlapWithId` in the figure. The function `ovWithId(xs, ys)` returns the overlapping same-category events in `xs` and `ys`. It has time complexity $O(|\texttt{xs}| + k|\texttt{ys}|)$ if each event in `ys` overlaps fewer than k same-category events in `xs`, as per the time complexity of Synchrony generator.

The direct translation of `ovWithId(xs, ys)` into SQL is given as `Query1` in Figure 6. Notice that it does not meet the syntactic requirement of a band join. So, a relational database system has to execute it using nested loops, resulting in $O(|\texttt{xs}| \cdot |\texttt{ys}|)$ time complexity. If

the relational database system supports a "single inequality" variant of the merge join, it can cut the time complexity by half; but this is still quadratic.

As `start` $<$ `end` holds for any event, it is can be shown that `overlapWithId(y, x)` if and only if `y.id` $=$ `x.id` and either `y.start` $<$ `x.start` $<$ `y.end` or `x.start` $\leq$ `y.start` $<$ `x.end`. So, an alternative SQL query can use the "union of two band joins" idea of Dignoes *et al.* (2021) to implement the same-category event-overlap function. This is `Query2` in Figure 6.

While there are different implementations of band join, their time complexity is lower bounded by output size. Thus, optimistically, the time complexity of each of the two band join is $O(|\texttt{xs}| + k|\texttt{ys}|)$ if each event in `ys` overlaps fewer than k events in `xs`. As there are two band joins and one union, the time complexity is $O(2|\texttt{xs}| + 2k|\texttt{ys}|)$, assuming the result of the second band join is directly concatenated to the first. Some implementations of band join do not support equality predicate; e.g., Dignoes *et al.* (2021) had to modify Postgres to make its band join support an equality predicate. In this case, `xs` and `ys` have to be re-sorted using only their `start` and `end` attributes and the selectivity k' must now include overlaps of events in different categories (so, $k' > k$.) Then the time complexity becomes worse.

It is worth remarking that, as demonstrated by Dignoes *et al.* (2021), the "union of two band joins" idea is the current state of art in implementing interval join in relational database systems research. The Synchrony generator implementation `ovWithId(xs, ys)` has time complexity $O(|\texttt{xs}| + k|\texttt{ys}|)$ when the selectivity is k, which compares favourably to $O(2|\texttt{xs}| + 2k|\texttt{ys}|)$. Importantly, it works directly on the `overlapWithId(y, x)` predicate (and any other antimonotonic predicates); whereas, for a relational database system, a user-programmer has to be skilled enough to recast an interval join to the more optimizer-friendly "union of two band joins." Another useful virtue is that the result of `syncGen(xs, ys)` is in the same order as `xs`, while the result produced by the "union of two joins" has lost this ordering. Thus, if the result is to be used as an input to a subsequent query (see the arranging-meeting example in Figure 7), the former might be usable directly; whereas, the latter might require extra sorting effort.

5.3 Synchronized iteration on multiple collections

Synchrony fold and derivatives described earlier are synchronizing iterations on two collections. How about synchronizing iterations on three or more collections using these functions? Consider a user-programmer writing a program `mtg0(ws, xs, ys, zs)` for finding the common overlaps between four collections of events. If `ws`, `xs`, `ys`, and `zs` are the available time slots of four people, then `mtg0(ws, xs, ys, zs)` are the time slots they are available to meet together.

A naive definition for `mtg0` in comprehension syntax, aiming at clarity, is given first in Figure 7. While `mtg0` is easy to understand, its quartic time complexity begs for improvement. An quick improvement is to insert some `overlap` predicates to eliminate nonoverlapping time slots as early as possible, as done by `mtg1` in Figure 7. If each available time slot of a person overlaps fewer than k time slots of another person, the time complexity of `mtg1` is quadratic, viz. $O(|\texttt{ws}|(|\texttt{xs}| + k|\texttt{ys}| + k^2|\texttt{zs}| + k^3))$. This is still not very efficient. So, `mtg2` in Figure 7 is an attempt using Synchrony generator to obtain a more efficient implementation. It also makes use of a nice idea on parallel comprehension-cum-monadic zip (Gibbons, 2016).

```
def mtg0(ws: Vec[Event], xs: Vec[Event], ys: Vec[Event], zs: Vec[Event]
): Vec[Event] =
  for (
    w <- ws; x <- xs; y <- ys; z <- zs;
    s = max(w.start, x.start, y.start, z.start);
    e = min(w.end, x.end, y.end, z.end);
    if s < e
  ) yield Event(start = s, end = e, id = w.id + x.id + y.id + z.id)


def mtg1(ws: Vec[Event], xs: Vec[Event], ys: Vec[Event], zs: Vec[Event]
): Vec[Event] =
  for (
    w <- ws;
    x <- xs; if overlap(x, w);
    y <- ys; if overlap(y, w);
    z <- zs; if overlap(z, w);
    s = max(w.start, x.start, y.start, z.start);
    e = min(w.end, x.end, y.end, z.end);
    if s < e
  ) yield Event(start = s, end = e, id = w.id + x.id + y.id + z.id)


def mtg2(ws: Vec[Event], xs: Vec[Event], ys: Vec[Event], zs: Vec[Event]
): Vec[Event] = {
  // Requires: ws,xs, ys, and zs sorted by (start, end).
  // Issue: The first four lines of codes below
  //        breaks the first-order restriction.
  val wxss  = syncGenGrp(isBefore, overlap)(ws, xs)
  val wyss  = syncGenGrp(isBefore, overlap)(ws, ys)
  val wzss  = syncGenGrp(isBefore, overlap)(ws, zs)
  val wxyzs = zip3(wxss, wyss, wzss)

  for (
    (wxs, wys, wzs) <- wxyzs;
    (w , xss) = wxs;
    (_,  yss) = wys;
    (_,  zss) = wzs;
    x <- xss; y <- yss; z <- zss;
    s = max(w.start, x.start, y.start, z.start);
    e = min(w.end, x.end, y.end, z.end);
    if s < e
  ) yield Event(start = s, end = e, id = w.id + x.id + y.id + z.id)
}
```

Fig. 7. The arranging-meeting example.

Assuming `ws`, `xs`, `ys`, and `zs` are sorted lexicographically based on `start` and `end` point of events, and all events overlap fewer than k other events, the time complexity of `mtg2(ws, xs, ys, zs)` is linear, $O(|\texttt{ws}| + 2k|\texttt{xs}| + |\texttt{ws}| + 2k|\texttt{ys}| + |\texttt{ws}| + 2k|\texttt{zs}| + 2|\texttt{ws}| + k^3|\texttt{ws}|) = O((k^3+5)|\texttt{ws}| + 2k(|\texttt{xs}| + |\texttt{ys}| + |\texttt{zs}|))$. Note that the $5|\texttt{ws}|$ overheads are due to (1) zipping `wxss`, `wyss`, and `wzss`; (2) scanning `wxyzs`; and (3) scanning `ws` three times when synchronizing with `xs`, `ys`, and `zs`. Nonetheless, this is much better than the quartic time complexity of `mtg0` and quadratic time complexity of `mtg1`, albeit it will be further improved in the next Section when Synchrony iterator is introduced.

Associated with the $5|\texttt{ws}|$ overheads is also the need to construct and store the intermediate collections `wxss`, `wyss`, `wzss`, and `wxyzs`, as Scala constructs these eagerly. Moreover, `wxss`, `wyss`, `wzss`, and `wxyzs` are nested collections; this breaks the first-order restriction. In

addition, the need to scan `ws` three times may be an issue when `ws` is a large data stream, as it implies needing to buffer the whole stream in memory. In a lazy programming language, this issue may go away, depending on how clever its garbage collector is.

Another somewhat unsatisfactory issue is the need for defining the function `zip3` to combine the three sequences of synchronizations of `xs`, `ys`, and `zs` to `ws`. We already know from the limited-mixing lemmas of Section 3 that `zip` is not efficiently definable under the first-order restriction, even though it is a straightforward two-line recursive function. And when the calendars of more people have to be synchronized, a zoo of `zip4`, `zip5`, etc. have to be written as well.[7] This issue is attributable to Scala's unsophisticated treatment of comprehension syntax. In a programming language (e.g., Haskell) which has more powerful ways to compile comprehension syntax using alternative binding semantics as well as enhancements to comprehension syntax design (Marlow *et al.*, 2016; Wadler & Peyton Jones, 2007; Gibbons, 2016; Lindley *et al.*, 2011), this issue of breaking the first-order restriction will likely disappear, though the $5|\texttt{ws}|$ time-complexity overheads highlighted earlier will likely remain.

6 Synchrony iterator

Recall again the motivating example from Figure 1, `ov1(xs, ys) = for (x <- xs; y <- ys; if overlap(x, y)) yield (x, y)`. Besides its poor quadratic time complexity which has already been highlighted, it suffers from another problem. If `Vec[·]` is a streaming data type, i.e. `xs` and `ys` are dynamic data generated continuously as events in them take place, then `ov1(xs, ys)` has to buffer all of `ys` and cannot move on to the second item in `xs` until the data stream `ys` is finished.

Our `syncFold` and `syncFoldGrp`—and thus, `syncMap`, `syncFlatMap`, `syncGen`, and `syncGenGrp`—do not suffer this same problem because, by Antimonotonicity Condition 2, they can move on to the next item in `xs` as soon as the current item in `ys` is after the current item in `xs` and cannot see the item. So, Synchrony fold and its derivatives do not have to buffer for all of `ys`. Nonetheless, the definitions of Synchrony fold in Section 4.1 and 4.2 do not produce any output until all items in `xs` and `ys` have been processed. As the actual processing by Synchrony fold only needs to see a small chunk of the two data streams at a time to compute the result for this small chunk, it is desirable to be able to return results incrementally in an on-demand manner.

A possible solution is using the relationship between `foldLeft` and `foldRight` to derive, from `syncGenGrp`, a lazy version `syncGenGrpLazy` where its result type is a `LazyList`.[8] While this is sufficient for getting a streaming version of Synchrony generator, `syncGenGrpLazy` has similar issues as `syncGenGrp` when it comes to synchronizing multiple collections. The

[7] The reader may find this `zip` issue confusing. Are we not already using recursion and other features when we define Synchrony fold and generator? Why are we complaining about having to define `zip3` in `mtg2`? Recall, in this paper, we separate an implementer-programmer who implements programming constructs and library functions from a user-programmer who uses these. The former has access to all features of Scala. The latter, in the context of this paper, is restricted to first-order Scala plus specifically permitted library functions which the former provides. As `mtg2` is an example of how to use Synchrony generator, it is expected to be written by the user-programmer. The user-programmer, being restricted to first-order Scala, thus cannot define an efficient `zip`. So, this user-programmer will have to write a much clumsier-looking program than `mtg2` for efficiency's sake; the clumsier-looking but efficient program is such an eyesore that we decided to omit it from this paper.

[8] In Scala, items in a `LazyList` are computed only when they are needed and are memoized.

arranging-meeting example of Figure 7, for instance, would have exactly the same implementation using `syncGenGrpLazy` as the version using `syncGenGrp`, viz. `mtg2`, but with every occurence of `syncGenGrp` replaced by `syncGenGrpLazy`. In particular, a user-programmer implementing it would also be required to write the functions for `LazyList` version of $\texttt{zip}_3$, $\texttt{zip}_4$, etc. depending on the number of people required for the meeting, as a generic `zip` function for zipping an arbitrary number of collections of arbitrary different types cannot be assigned a valid type in current strongly typed programming languages.

As another mechanism for incrementally producing items on demand, an iterator comes to mind. A normal Scala iterator `yi` provides a `yi.next()` method which on-the-fly computes and produces the next item in the iteration. Although this is simple, we decided against it. The reason is `yi.next()` is iterating only on one collection. A user-programmer would thus be forced to organize the synchronization with the other collections using some additional mechanism, and we would be back to square one.

These two problems—viz. streaming and multi-collection synchronized iteration—are addressed in this section. In particular, *Synchrony iterator* is conceived in this section. A Synchrony iterator `yi = new EIterator(ys, isBefore, canSee)` provides a `yi.syncedWith(x)` method that on-the-fly computes and produces the items in the iteration on `ys` that should be synchronized to (i.e. can see) the item `x`, under the assumption that successive invocations of `syncedWith` are given, as the values of `x`, successive items of a collection `xs` ordered such that `isBefore` is monotonic with respect to `(xs, ys)` and `canSee` is antimonotonic with respect to `isBefore`.

This design of Synchrony iterator has two advantages over the standard iterator. Firstly, a nice byproduct of this design is that the same `x` can be used to synchronize multiple Synchrony iterators simultaneously. Using this alternative way to express multi-collection synchronized iteration avoids the `zip` issue mentioned in the discussion on `mtg2`. Secondly, like iterators in general, Synchrony iterator requires side effects. However, unlike the standard iterator, some safe-use conditions can be provided on Synchrony iterator. These conditions isolate these side effects and are sufficient for restoring transparent equational reasoning for programs involving Synchrony iterator.

6.1 Designing Synchrony iterator

Deriving a version of Synchrony generator that incrementally computes and returns its result is a fairly typical programming problem. So, we give it a try first by looking at the definition of `syncGenGrp`. The codes for the Synchrony generator `syncGenGrp`, after unfolding through `synGenGrp(bf, cs)(xs, ys)` = `syncFoldGrp((x, zs, a) => a :+ (x, zs), Vec(), bf, cs)(xs, ys)`, are shown in the top half of Figure 8.

It is quite apparent that a simple rearrangement of the `aux` function used in defining `syncGenGrp` is sufficient to make it return one element of the result at a time. This is shown in the bottom half of Figure 8. In this rearrangement, the `EIterator` class is introduced. Objects of this class are called *eiterators* (pronounced "iterators.") An eiterator `yi = new EIterator(ys, isBefore, canSee)` can be regarded as an *enhanced* iterator on the collection `ys`. The eiterator is characterized by a method `yi.syncedWith(x)`, which is the rearranged `aux` function from `syncGenGrp`.

```
// The codes for syncGenGrp after unfolding through
//   synGenGrp(bf, cs)(xs, ys) =
//      syncFoldGrp((x, zs, a) => a :+ (x, zs), Vec(), bf, cs)(xs, ys).
def syncGenGrp[A,B]
  (bf: (B,A) => Boolean, cs: (B,A) => Boolean)
  (xs: Vec[A], ys: Vec[B])
: Vec[(A,Vec[B])] = {

  def aux(xs: Vec[A], ys: Vec[B], zs: Vec[B], acc: Vec[(A,Vec[B])])
  : Vec[(A,Vec[B])] = {
    if (xs.isEmpty) acc
    else if (ys.isEmpty && zs.isEmpty) acc
    else if (ys.isEmpty) aux(xs.tail, zs, Vec(), acc :+ (xs.head, zs))
    else {
      val (x,y) = (xs.head, ys.head)
      (bf(y, x), cs(y, x)) match {
        case (true, false) => aux(xs, ys.tail, zs, acc)
        case (false, false) => aux(xs.tail, zs ++: ys, Vec(), acc :+ (x,zs))
        case (_, true) => aux(xs, ys.tail, zs :+ y, acc)
      }
    }
  }

  aux(xs, ys, Vec(), Vec())
}


// Rearranging syncGenGrp's aux function to return one element
// of the result at a time. This provides a preliminary
// implementation of Synchrony iterator.
class EIterator[A,B](
  elems: Vec[B],
  bf: (B,A)=>Boolean, cs:(B,A)=>Boolean) {

  private var es = elems

  def syncedWith(x: A): Vec[B] = {
    def aux(zs: Vec[B]): Vec[B] = {
      if (es.isEmpty && zs.isEmpty)  zs
      else if (es.isEmpty) { es = zs; zs }
      else {
        val y = es.head
        (bf(y, x), cs(y, x)) match {
          case (true, false) => { es = es.tail; aux(zs) }
          case (false, false) => { es = zs ++: es; zs }
          case (_, true) => { es = es.tail; aux(zs :+ y) }
        }
      }
    }
    aux(Vec())
  }
}
```

Fig. 8. Preliminary definition of `EIterator`, shown along side the unfolded definition of `syncGenGrp`. The `syncedWith` method of the former is derived from the `aux` function of the latter.

The theorem below shows that when `yi = new EIterator(ys, isBefore, canSee)` is a fresh eiterator on `ys`, calling `yi.syncedWith(x)` on each successive item `x` in `xs`, returns the corresponding successive item `(x, zs)` in `syncGenGrp(isBefore, canSee)(xs, ys)`. Furthermore, the total time complexity is the same. Thus, an eiterator generates—at the same efficiency—the same items produced by a Synchrony generator, and it produces these

items one at a time when its `syncedWith` method is called iteratively. For this reason, an eiterator is called a *Synchrony iterator*.

Theorem 6.1. *Suppose* `isBefore` *is monotonic with respect to* `(xs, ys)`*, and* `canSee` *is antimonotonic with respect to* `isBefore`*. Then, the following two programs define the same function.*

1. `syncGenGrp(isBefore, canSee)(xs, ys)`
2. ```
   val yi = new EIterator(ys, isBefore, canSee)
   for (x <- xs; zs <- yi.syncedWith(x)) yield (x, zs)
   ```

Both programs have time complexity $O(|\texttt{xs}| + 2k|\texttt{ys}|)$*, assuming* `isBefore` *and* `canSee` *have constant time complexity and each item in* `ys` *can see fewer than* k *items in* `xs`*.*

Proof When an eiterator `yi = new EIterator(ys, isBefore, canSee)` is freshly created, its internal variable `es` is initialized to the collection `ys`. Let the items in `xs` be $\texttt{x}_1$, ..., $\texttt{x}_n$, in this order. Suppose `isBefore` is monotonic with respect to `(xs, ys)`, and `canSee` is antimonotonic with respect to `isBefore`. Suppose `yi.syncedWith(`$\texttt{x}_1$`)`, ..., `yi.syncedWith(`$\texttt{x}_n$`)` are called in this order. Let $\texttt{zs}_1$, ..., $\texttt{zs}_n$ be the corresponding results. Let $\texttt{es}_1$, ..., $\texttt{es}_n$ be the value of the internal variable `es` at the end of each of these calls. And let $\texttt{e}_0 = \texttt{ys}$.

By construction, for each `y` in `ys`, such that `isBefore(y, `$\texttt{x}_j$`)`, it is the case `canSee(y, `$\texttt{x}_j$`)` if and only if `y` is in $\texttt{zs}_j$ and $\texttt{es}_j$. Also, by construction, for each `y` in `ys`, such that not `isBefore(y, `$\texttt{x}_j$`)`, it is the case that `y` is in $\texttt{es}_j$; and by Antimonotonicity Condition 2, `y` is in $\texttt{zs}_j$ if and only if `canSee(y, `$\texttt{x}_j$`)`. Thus, for `y` in `ys`, `y` is in $\texttt{zs}_j$ if and only if `canSee(y, `$\texttt{x}_j$`)`. So,

$\texttt{zs}_j =$ `for (y <- ys; if canSee(y, `$\texttt{x}_j$`)) yield y`

`Vec((`$\texttt{x}_j$`, `$\texttt{zs}_j$`)) = syncGenGrp(isBefore, canSee)(Vec(`$\texttt{x}_j$`), ys)`

Then, the first part of the theorem follows from Corollary 4.6,

```
  syncGenGrp(isBefore, canSee)(xs, ys)
= for (x <- xs; zs <- yi.syncedWith(x)) yield (x, zs)
```

Looking at the definition of the function `aux` in `syncedWith`, when processing `yi.syncedWith(`$\texttt{x}_j$`)`, each time `aux` is called, it reduces the number of items in $\texttt{es}_{j-1}$ (and thus `ys`) by 1; or it increases the number of items by $|\texttt{zs}_j|$ exactly once, when it returns. As mentioned earlier, the items in $\texttt{zs}_j$ are those `y` that can see $\texttt{x}_j$. Thus, the total number of times `aux` gets called when processing `yi.syncedWith(`$\texttt{x}_1$`)`, ..., `yi.syncedWith(`$\texttt{x}_n$`)`, is $|\texttt{ys}| + \sum_j |\texttt{zs}_j|$. By assumption of the theorem, each item in `ys` can see fewer than k items in `xs`. So, each item in `y` appears in fewer then k distinct $\texttt{zs}_j$. Thus, $\sum_j |\texttt{zs}_j| < k|\texttt{ys}|$. Also, the prepend operator `zs ++: es` is linear in $|\texttt{zs}|$; these add an overhead of $\sum_j |\texttt{zs}_j| < k|\texttt{ys}|$. So, `for (x <- xs; zs <- yi.syncedWith(x)) yield (x, zs)` has time complexity $O(|\texttt{xs}| + 2k|\texttt{ys}|)$. This proves the second part of the theorem. ■

The following useful details can also be extracted from the proof above.

Proposition 6.2. *Suppose* `isBefore` *is monotonic with respect to* `(xs, ys)`*, and* `canSee` *is antimonotonic with respect to* `isBefore`*. Let the eiterator* `yi = new EIterator(ys, isBefore, canSee)` *be freshly created. Let* $\texttt{x}_1$, ..., $\texttt{x}_n$ *be some*
```

```
def mtg3(
  ws: Vec[Event], xs: Vec[Event], ys: Vec[Event], zs: Vec[Event]
): Vec[Event] =  {
  // Requires: ws, xs, ys, zs sorted lexicographically by (start, end).
  // Note: isBefore and overlap are as defined in Figure 1.
  val xi = new EIterator(xs, isBefore, overlap);
  val yi = new EIterator(ys, isBefore, overlap);
  val zi = new EIterator(zs, isBefore, overlap);
  for (
    w <- ws;
    x <- xi.syncedWith(w);
    y <- yi.syncedWith(w);
    z <- zi.syncedWith(w);
    s = max(w.start, x.start, y.start, z.start);
    e = min(w.end, x.end, y.end, z.end);
    if s < e
  ) yield Event(start = s, end = e, id = w.id + x.id + y.id + z.id)
}
```

Fig. 9. The arranging-meeting example expressed using Synchrony iterator.

of the items in xs*, with possible repetitions and omissions of items in* xs*, such that for* $1 \leq j < j' \leq n$ *where* $\mathtt{x}_j \neq \mathtt{x}_{j'}$*, it is the case that* $(\mathtt{x}_j \ll \mathtt{x}_{j'} \mid \mathtt{xs})$*. Suppose* $\mathtt{yi.syncedWith}(\mathtt{x}_j)$ *is called in sequence for each* $\mathtt{x}_j$*. Let* $\mathtt{zs}_j$ *be the corresponding result and* $\mathtt{es}_j$ *be the value of the internal variable* es *of the eiterator* yi *at the end of each of these calls. And let* $\mathtt{e}_0 = \mathtt{ys}$*. Then,*

$$\begin{aligned} \mathtt{zs}_j &= \{\ \mathtt{new\ EIterator(es}_{j-1}\mathtt{,\ isBefore,\ canSee)}\ \}\mathtt{.syncedWith(x}_j\mathtt{)} \\ &= \{\ \mathtt{new\ EIterator(ys,\ isBefore,\ canSee)}\ \}\mathtt{.syncedWith(x}_j\mathtt{)} \\ &= \mathtt{for\ (y\ <-\ ys;\ if\ canSee(y,\ x}_j\mathtt{))\ yield\ y} \end{aligned}$$

That is, only the ordering of $\mathtt{x}_j$ *matters when calling* syncedWith*; repetitions and omissions of items in* xs *have no impact.*

The design of Synchrony iterator thus meets the objective of incrementally computing and producing synchronized items in a collection ys to items in a collection xs.

Fortuitously, the synchronization provided by Synchrony iterator is specified via yi.syncedWith(x); i.e., xs does not need to be given as part of the specification. This design facilitates the simultaneous synchronization of items in multiple collections to the same item x. In particular, let $\mathtt{yi}_1$, ..., $\mathtt{yi}_n$ be eiterators on the n collections $\mathtt{ys}_1$, ..., $\mathtt{ys}_n$ that are to be synchronized to items in xs. Then for each item x in xs, the methods $\mathtt{yi}_1$.syncedWith(x), ..., $\mathtt{yi}_n$.syncedWith(x) are called to achieve simultaneous synchronized iteration on the n collections to the collection xs, like this:

```
val yi_1 = new EIterator(ys_1, bf_1 cs_1); ...;  val yi_n = new EIterator(ys_n, bf_n cs_n);
for (x <- xs; y_1 <- yi_1.syncedWith(x); ...; y_n <- yi_n.syncedWith(x)) yield ...
```

The function mtg3 in Figure 9, which revisits the arranging-meeting example from Section 5.3, illustrates this simultaneous synchronization. Notice that mtg3 is first order. And, in contrast to the approach adopted earlier by mtg2 from Figure 7, mtg3 dispenses with the need to define a zoo of zip's to structure synchronized iteration in multiple collections.

However, the time complexity of mtg3 may not be as good as mtg2. Suppose all events overlap fewer than k other events. The time complexity of mtg3 is $O(|\mathtt{ws}| +$

$2k|\texttt{xs}| + 2k^2|\texttt{ys}| + 2k^3|\texttt{zs}| + k^3|\texttt{ws}|)$, because `x <- xi.syncedWith(w)` is called once for each `w`, `y <- yi.syncedWith(w)` is called once for each `x`, and `z <- zi.syncedWith(w)` is called once for each `y`.

Fortunately, although `y <- yi.syncedWith(w)` and `z <- zi.syncedWith(w)` are called multiple times for different `x`'s and `y`'s respectively, these calls depend on `w` and not on `x` and `y`. So, the problem is easy to solve by making `syncedWith` remember its immediate last result. Figure 10 shows the revised `EIterator` class incorporating this simple solution. We have so far used `Vec[·]` to denote a collection type. Perhaps this gives the appearance that `xs` and `ys` are collection types of the same kind, i.e., both are vectors, both are lists, etc. Actually, this does not need to be the case. So, we show this in this revised version of `EIterator` as well. Specifically, `yi = new EIterator(ys, bf, cs)` now constructs an eiterator for any kind of iterable object `ys`, e.g., a `LazyList[B]`, which is Scala's preferred data type for representing data streams. Also, the method `yi.syncedWith(x)` now returns a `List[B]`. And the collection `xs`, where `x` comes from, can be yet another kind of collection type, e.g., a `Vector[A]`. Incidentally, the prepend `++:` and postpend `:+` operations on vectors, are not needed in this version of `EIterator`.

With this simple modification to Synchrony iterator, the time complexity of `mtg3` becomes $O((k^3+1)|\texttt{ws}| + 2k(|\texttt{xs}| + |\texttt{ys}| + |\texttt{zs}|))$. Now, `mtg3` is even more efficient than `mtg2`, successfully reducing the latter's $5|\texttt{ws}|$ overheads to $|\texttt{ws}|$, as well as avoiding the construction of several large intermediate collections. Moreover, even when `ws`, `xs`, `ys`, and `zs` are large dynamic data streams, `mtg3` can produce their common time slots incrementally as overlapping events arrive.

It is worth diving deeper into the details of the revised definition of `EIterator` in Figure 10. `EIterator` memoizes the previous result in `ores` and the previous value of `x` in `ox`. If the next value of `x` is same as the one memoized earlier in `ox`, the result memoized earlier in `ores` is returned immediately. Otherwise, the synchronized iteration resumes from `ores` and continues onward to `es`. This actually kills a second bird with the same stone: In the earlier definition of Synchrony iterator in Figure 8, when both `bf(y, x)` and `cs(y, x)` are false, `zs` must be prepended back to `es` before returning `zs` as the result. This prepending step is dispensed with in this revised definition of `EIterator` as the result is now already memoized in `ores` and the iteration in response to the next call value `x` resumes from `ores` before continuing onward to `es`.

Under the hood in Scala, being a `List[·]`, `ores` is a "boxed value"; i.e., it is a pointer. Thus, if there are multiple consecutive `x`'s which have the same value, the corresponding `yi.syncedWith(x)` results are exactly the same pointer. This has a rather nice practical implication, akin to factorized databases (Olteanu & Schleich, 2016). As an illustration, let the collection `xs` be just a sequence repeating the same value `u`, and the collection `ys` be just a sequence repeating the same value `v`. Suppose also that `cs(v, u)` is true. Then, it does not matter whether `bf(v, u)` is true, `for (x <- xs; zs = yi.syncedWith(x)) yield (x, zs)` has linear physical size $O(|\texttt{xs} + \texttt{ys}|)$, even though—semantically—there are $|\texttt{xs}| \cdot |\texttt{ys}|$ number of items from `xs` and `ys` in it. Although not explored here, this property may be further exploited for designing more efficient algorithms, e.g., for database query processing, perhaps in the manner of Henglein & Larsen (2010) and Olteanu & Schleich (2016).

Also, in practice, `isBefore(y, x)` and `canSee(y, x)` predicates do not use all the information in `y` and `x`. In fact, they often have a form like `bf(y, x) = bfk(`ψ`(y), `ϕ`(x))` and

```
class EIterator[A,B](
  elems: Iterable[B],
  bf: (B,A)=>Boolean, cs: (B,A)=>Boolean)
{
  private var es: Iterable[B] = elems
  private var ores: List[B]   = List() // last result
  private var ox: Option[A]   = None   // last x

  // When iterating, use items in ores before items in es.
  private def empty = es.isEmpty && ores.isEmpty
  private def hd    = if (ores.isEmpty) es.head else ores.head
  private def nx()  = if (ores.isEmpty) { es = es.tail }
                      else { ores = ores.tail }

  def syncedWith(x: A): List[B] = {
    def aux(zs: List[B]): List[B] =
      if (empty) { zs }
      else {
        val y = hd
        (bf(y, x), cs(y, x)) match {
          case (true, false) => { nx(); aux(zs) }
          case (false, false) => { zs }
          case (_, true) => { nx(); aux(y +: zs) }
        }
      }
    // Use the last result if this x is same as the last x
    if (ox == Some(x)) { ores }
    else { ox = Some(x); ores = aux(List()).reverse; ores }
  }
}


class EIteratorWithKey[KA,KB,A,B](
  keya: A => KA, keyb: B => KB,
  elems: Iterable[B],
  bfk: (KB,KA)=>Boolean, csk: (KB,KA)=>Boolean)
extends EIterator[A,B](null, null, null)
{
  // EIterator ek for synchronizing elems to keya(x) instead of x.
  private val ek: EIterator[KA,B] = {
    val bf = (y: B, kx: KA) => bfk(keyb(y), kx)
    val cs = (y: B, kx: KA) => csk(keyb(y), kx)
    new EIterator(elems, bf, cs)
  }

  // Override syncedWith(x) by ek.syncedWith(keya(x)).
  // This is equivalent to defining the isBefore and canSee
  // predicates for EIteratorWithKey as:
  //     bf(y, x) = bfk(keyb(y), keya(x))
  //     cs(y, x) = csk(keyb(y), keya(x))
  override def syncedWith(x: A): List[B] = ek.syncedWith(keya(x))
}
```

Fig. 10. Revised definition of Synchrony iterator `EIterator` and its derivative `EIteratorWithKey`, whose `isBefore` predicate (`bfk`) and `canSee` predicate (`csk`) are defined using sorting keys (`keya`, `keyb`).

`cs(y, x) = csk(`ψ`(y), `ϕ`(x))`, where $\psi(\cdot)$ and $\phi(\cdot)$ are some sorting keys of `ys` and `xs` respectively. As `xs` is sorted by $\phi(\cdot)$, for any $(x_i \ll x_j \mid \texttt{xs})$ where $\phi(x_i) = \phi(x_j)$, it is the case that $\phi(x_i) = \phi(x_k) = \phi(x_j)$ for all $(x_i \ll x_k \ll x_j \mid \texttt{xs})$; this is so even when $x_i \neq x_k \neq x_j$. This means `yi.syncedWith(`x_i`)` = `yi.syncedWith(`x_k`)` = `yi.syncedWith(`x_j`)`,

assuming `yi.syncedWith(x`$_i$`)`, `yi.syncedWith(x`$_k$`)`, and `yi.syncedWith(x`$_j$`)` are called in this sequence. However, when $\texttt{x}_i \neq \texttt{x}_k \neq \texttt{x}_j$, `yi.syncedWith(x`$_i$`)`, `yi.syncedWith(x`$_k$`)`, and `yi.syncedWith(x`$_j$`)` would be pointers to three separate physical lists comprising exactly the same sequence of items from `ys`. To avoid this situation, instead of memoizing the argument `x`, the `syncedWith` method of `EIterator` should memoize $\phi(\texttt{x})$. In Scala, this can be accomplished by defining a subclass `EIteratorWithKey` of `EIterator`, where `EIteratorWithKey` redefines `syncedWith(x)` to `syncedWith(`ϕ`(x))`, as shown in Figure 10. Then, instead of creating an eiterator by `yi = new EIterator(ys, bf, cs)`, it can be created as `yi = new EIteratorWithKey(`$\phi(\cdot)$`, `$\psi(\cdot)$`, ys, bfk, csk)`.

6.2 Safe use of Synchrony iterator

Synchrony iterator is defined using side effects. Each time `syncedWith` is invoked on an eiterator, the local variable `es` and its local result cache `ores` and `ox` are updated. This can make a program difficult to understand when Synchrony iterator is used in an undisciplined way. Therefore, the following conditions are imposed to ensure better discipline in using Synchrony iterator. To specify these conditions, the notation $\mathscr{F}[\cdot]$ denotes an expression with a "hole"—called a "context"—and $\mathscr{F}[e]$ denotes the same expression but with the expression e substituted into the hole.

Definition 6.3 (Safe-use)**.** *The following conditions are presumed to hold on a program for each expression* `yi.syncedWith(x)` *that appears in the program.*

1. *There is a collection* `xs`*, and* `x` *takes successive values in* `xs`*. That is, the expression* `yi.syncedWith(x)` *appears in an enclosing expression that binds* `x` *to the collection* `xs`*. In general, the enclosing expression* $\mathscr{C}$`[yi.syncedWith(x)]` *looks like, or gets desugared into, one of these forms:*

   ```
   xs flatMap (x => 𝓕[yi.syncedWith(x)])

   xs map (x => 𝓕[yi.syncedWith(x)])

   xs filter (x => 𝓕[yi.syncedWith(x)])
   ```

2. `yi` *is an eiterator on some collection* `ys`*.*
3. `isBefore` *is monotonic with respect to* (`xs`, `ys`)*.*
4. `canSee` *is antimonotonic with respect to* `isBefore`*.*
5. `yi.syncedWith(x)` *produces the same value as* `ys filter (y => canSee(y, x))`*, though not necessarily with the same efficiency, in the context of this program. That is,* $\mathscr{C}$`[yi.syncedWith(x)]` $=$ $\mathscr{C}$`[ys filter (y => canSee(y, x))]`*.*

It may seem onerous to programmers to have these conditions imposed on them. In reality, they only need to take responsibility for Safe-use Conditions 3 and 4, as these are non-trivial for the compiler to verify automatically in some cases; nonetheless, they are often easy to achieve. The other safe-use conditions are easy for a compiler to check or to enforce in pragmatically, as explained below, or to train programmers to comply with.

Safe-use Condition 1 is trivial, and can be easily checked and enforced by the compiler. It simply says a Synchrony iterator on a collection `ys` should always be used inside the scope of the generator that `ys` is synchronized to.

Safe-use Condition 2 is also trivial. It is just standard type checking.

Safe-use Condition 5, though seems non-trivial at first sight, can be achieved in a pragmatic way which can be enforced by the compiler. In fact, only two basic rules are needed. First, if there is another expression `yi.syncedWith(x′)` on the same eiterator `yi`, we must have `x == x′`. That is, all occurrences of the eiterator `yi` are identical; i.e. synchronized to the same `x` in `xs`. Or, better still, insist on `yi` to occur only twice in the program, once when the eiterator is being created (i.e. `yi = new EIterator(ys, isBefore, canSee)`), and once when the eiterator is being used for the only time (i.e. `yi.syncedWith(x)`). Second, the eiterator `yi` should be constructed immediately before the generator of `xs`. That is, programmers should always use `yi.syncedWith(x)` inside an enclosing expression that looks like, or gets desugared to, one of these forms:

```
val yi = new EIterator(ys, isBefore, canSee)
xs flatMap (x => ℱ[yi.syncedWith(x)])

val yi = new EIterator(ys, isBefore, canSee)
xs map (x => ℱ[yi.syncedWith(x)])

val yi = new EIterator(ys, isBefore, canSee)
xs filter (x => ℱ[yi.syncedWith(x)])
```

Even though eiterators have side effects that change their state (viz. their local variables `es`, `ores`, and `ox`), the two rules under Safe-use Condition 5 isolate these side effects. Without loss of generality, by the second rule, suppose an eiterator appears like this:

```
val yi = new EIterator(ys, isBefore, canSee)
xs flatMap (x => ℱ[yi.syncedWith(x)])
```

By the first rule, `yi` appears only in the exact form `yi.syncedWith(x)`, whose value depends only on `x`. Being in a comprehension, `x` takes successive values x_j of `xs`. So, by Proposition 6.2, it is guaranteed that `yi.syncedWith(`x_j`)` $=$ `ys filter (y => canSee(y, `x_j`))`. That is,

```
  { val yi = new EIterator(ys, isBefore, canSee);
    xs flatMap (x => ℱ[yi.syncedWith(x)]) }
= xs flatMap (x => ℱ[ys filter (y => if canSee(y, x))])
```

In other words, it permits the left-hand-side (which has side effects) to be replaced by the right-hand-side (which has no side effects) when one is reasoning extensionally. Thus, despite its side effects, under the safe-use conditions, one might justifiably claim that Synchrony iterator is a purer programming paradigm than a standard iterator.

Incidentally, the equivalence highlighted above also implies that Synchrony iterator, under the safe-use conditions, is a conservative extension of first-order restricted Scala sans library functions.

Theorem 6.4. *The extensional expressive power of Scala under the first-order restriction, is the same with or without Synchrony iterator under the safe-use conditions. However, more efficient algorithms for some functions (e.g., a linear-time algorithm for low-selectivity join) can be defined when Synchrony iterator is made available in this fragment of Scala.*

6.3 Referential transparency of Synchrony iterator

The Safe-use conditions of Synchrony iterator assure its referential transparency. A rather attractive implication is that equational reasoning that holds for standard collection types, but fails on standard iterators, holds for Synchrony iterator. This is a direct consequence of Safe-use Condition 5. We illustrate this with the equations for code motion, redundant-code elimination, and parallelism.

Code motion

The "code-motion" equation below is valid for standard collection types, provided the free variables of e_2 are a subset of the free variables of `u => `$\mathscr{F}[e_2]$, and e_2 has no observable side effects.

```
  e1 flatMap (u => 𝓕[e2])
= { val v = e2; e1 flatMap (u => 𝓕[v]) }
```

The code-motion equation is inapplicable to standard iterators. In contrast, it is applicable to Synchrony iterator under safe-use conditions. Specifically, the following holds:

```
  e1 flatMap (u => 𝓕[yi.syncedWith(x)])
= { val v = yi.syncedWith(x); e1 flatMap (u => 𝓕[v]) }
```

The validity of this code-motion equation is a consequence of Safe-use Condition 5. To wit, assume `yi = new EIterator(ys, bf, cs)` for some `ys`, `bf`, and `cs`, then proceed as follow.

```
  e1 flatMap (u => 𝓕[yi.syncedWith(x)])
= e1 flatMap (u => 𝓕[ys filter (y => cs(y, x))])
= { val v = ys filter (y => cs(y, x)); e1 flatMap (u => 𝓕[v]) }
= { val v = yi.syncedWith(x); e1 flatMap (u => 𝓕[v]) }
```

Redundant-code elimination

The "redundant-code elimination" equation below is valid for standard collection types, provided the expression `e` has no observable side effects.

```
  (e flatMap f) ++ (e flatMap g)
= { val v = e; (v flatMap f) ++ (v flatMap g) }
```

This redundant-code elimination equation is inapplicable when `e` is an expression having an iterator type. In contrast, under safe-use conditions, it is applicable to Synchrony iterator despite its having side effects. Specifically, the following holds:

```
  (yi.syncedWith(x) flatMap f) ++ (yi.syncedWith(x) flatMap g)
= { val v = yi.syncedWith(x); (v flatMap f) ++ (v flatMap g) }
```

The validity of this redundant-code elimination is again a consequence of Safe-use Condition 5. As before, assume `yi = new EIterator(ys, bf, cs)` for some `ys`, `bf`, and `cs`, and proceed as follow.

```
  (yi.syncedWith(x) flatMap f) ++ (yi.syncedWith(x) flatMap g)
= (ys filter (y => cs(y, x) flatMap f)) ++ (ys filter (y => cs(y, x) flatMap g))
= { val v = ys filter (y => cs(y, x)); (v flatMap f) ++ (v flatMap g) }
= { val v = yi.syncedWith(x); (v flatMap f) ++ (v flatMap g) }
```

Homomorphism over `flatMap`

The "homomorphism" equation below is valid for standard collection types, provided expressions `e`, `f`, and `g` have no observable side effects. This equation is the basis for parallelization of `flatMap` in, e.g., Hadoop-like platforms.

```
  (e ++ f) flatMap g
= (e flatMap g) ++ (f flatMap g)
```

A similar homomorphism equation holds for `syncedWith`. Suppose `val yi = new EIterator(us ++ vs, bf, cs)` for some `us`, `vs`, `bf`, and `cs`. Then, after replacing `val yi = new EIterator(us ++ vs, bf, cs)` by `{ val ui = new EIterator(us, bf, cs); val vi = new EIterator(vs, bf, cs) }`, the equation below holds.

```
  yi.syncedWith(x)
= ui.syncedWith(x) ++ vi.syncedWith(x)
```

This equation follows because

```
  yi.syncedWith(x)
= (us ++ vs) filter (y => cs(y, x))
= (us filter (y => cs(y, x))) ++ (vs filter (y => cs(y, x)))
= ui.syncedWith(x) ++ vi.syncedWith(x)
```

This equation offers a simple way to parallelize `syncedWith`.

6.4 Possible syntax for Synchrony iterator

Considering the safe-use conditions, it is perhaps pertinent to suggest a syntax for Synchrony iterator that automatically enforces all the safe-use conditions, apart from the safe-use conditions on monotonicity and antimonotonicity. One possibility is to introduce the following generator pattern into comprehension syntax:

```
(x, zs_1, ..., zs_n) <- xs syncWith(ys_1, bf_1, cs_1) ...
                           syncWith(ys_n, bf_n, cs_n)
```

This way, the `EIterator` class can be hidden from user-programmers, and they can be told that zs_j = `ys.filter((y) => cs`$_j$`(y, x))` at all times in terms of value, as per Proposition 6.2, but is obtained very efficiently.

This generator pattern is compiled by desugaring it to

```
(x, zs1, ..., zsn) <- {
     val yi1 = new EIterator(ys1, bf1, cs1); ...;
     val yin = new EIterator(ysn, bfn, csn)
     for (
        x <- xs;
        zs1 = yi1.syncedWith(x); ...;
        zsn = yin.syncedWith(x);
     ) yield (x, zs1, ..., zsn) }
```

Usual "deforestation" rules (Wadler, 1990) should be able to optimize this further to remove the intermediate collection introduced by this desugaring. If not, the generator pattern can also be desugared into the chain of generator and assignment patterns below:

```
yi1 = new EIterator(ys1, bf1, cs1); ...;
yin = new EIterator(ysn, bfn, csn)
x <- xs;
zs1 = yi1.syncedWith(x); ...;
zsn = yin.syncedWith(x);
```

The program `mtg4` in Figure 11 is a rewrite of the program `mtg3` from Figure 9 using this suggested syntax for Synchrony iterator. As can be seen, using this syntax, the three Synchrony iterators `xi`, `yi`, and `zi` that earlier appeared explicitly in `mtg3` are now tucked away from sight. The user-programmer is thus presented with a pure functional comprehension syntax which uses a slightly enhanced generator form.[9]

7 Some use-cases and a stress test

Synchrony fold and Synchrony iterator for querying relational databases in general, genomic datasets in particular, and timestamped data streams is discussed here. A stress test on using them on genomic datasets is also presented.

7.1 Relational database queries

The use-case of Synchrony fold and Synchrony iterator in the context of relational database querying should be quite clear already. Further technical and theoretical details are given in a companion paper (Wong, 2021). So, here, we just point out that only one extra function is needed to make all relational database queries (including group-by, order-by, and aggregate functions) efficiently implementable in first-order restricted Scala endowed with

[9] This tantalizing syntax is used for illustrative purpose later in Section 7.4. However, in the rest of this work, we eschew using it in favour of the plain `yi = new EIterator(ys, bf, cs)` and `yi.syncedWith(x)` as our eiterator constructs. The idea and design of Synchrony fold, Synchrony generator, and Synchrony iterator are partly driven by our desire in suggesting a small set of library functions for general synchronized iteration. The `EIterator` and `EIteratorWithKey` classes and the functions defining Synchrony fold and Synchrony generator have the crucial advantage of being readily copied and adopted for a wide variety of programming languages without modifying any of their compilers. Whereas, introducing new syntax into any programming language faces the obstacle of modifying its compiler, which requires significantly more technical effort (perhaps also requires lots of lobbying); it is thus an unlikely scenario for most programming languages.

```
def mtg4(
  ws: Vec[Event], xs: Vec[Event], ys: Vec[Event], zs: Vec[Event]
): Vec[Event] =  {
  // Requires: ws, xs, ys, zs sorted lexicographically by (start, end).
  // Note: isBefore and overlap are as defined in Figure 1.
  for (
    (w, wxs, wys, wzs) <- ws syncWith(xs, isBefore, overlap)
                             syncWith(ys, isBefore, overlap)
                             syncWith(zs, isBefore, overlap)
    x <- wxs; y <- wys; z <- wzs;
    s = max(w.start, x.start, y.start, z.start);
    e = min(w.end, x.end, y.end, z.end);
    if s < e
  ) yield Event(start = s, end = e, id = w.id + x.id + y.id + z.id)
}
```

Fig. 11. The arranging-meeting example revisited again. The program `mtg4` is a rewrite of the program `mtg3` from Figure 9 using the generator syntax suggested for Synchrony iterator.

Synchrony fold and Synchrony iterator. That extra function is `sortWith(f)(xs)` which sorts the collection `xs: Vec[A]` using the ordering function `f: (A,A) => Boolean`. This is because Synchrony fold and Synchrony iterator require their input to be suitably sorted beforehand.

Actually, sorting at quadratic time complexity is already expressible in first-order restricted Scala endowed with Synchrony fold. Theorem 4.5 implies that all functions defined using first-order restricted Scala with Synchrony fold has time complexity of the form $O(m^n)$ where m is input size. Since efficient sorting requires $\Omega(m \log(m))$ time in general, this means sorting takes $\Theta(m^2)$ time in first-order-restricted Scala with Synchrony fold. Thus, it is necessary to provide an efficient `sortWith` sorting function to user-programmers, to ensure that they are able to implement any relational database queries efficiently in this framework.

It is worth remarking that on-disk sorting is much easier to implement than on-disk indexed tables; cf. Silberschatz *et al.* (2016). It is thus a virtue of Synchrony iterator, which needs only the former when processing very large collections, relative to approaches that try to compile join-expressing comprehensions into indexed tables.

7.2 Genometric queries

A second use-case is genometric queries on genomic datasets. BEDOPS (Neph *et al.*, 2012) and GMQL (Masseroli *et al.*, 2019) are two notable toolkits for processing these datasets, and support similar query operations. The former via unix-style commands. The latter via a specialized GenoMetric Query Language. The data model is highly constrained in such domain-specific toolkits. GMQL is used here for illustration, modulo some liberty taken with GMQL's syntax.

There are only a few main object types. The first main object type is the genomic region; this is `Bed(chrom: String, start: Int, end: Int, ...)` for a region located on chromosome `chrom`, beginning at position `start`, ending at position `end`, plus some other pieces of information which are omitted here. Regions on the same chromosome are ordered by their `start` and `end` point lexicographically; regions on different chromosomes are ordered by their `chrom` value. This ordering $<_{\texttt{Bed}}$ defines the default `isBefore` predicate on regions, viz. `isBefore(y, x)` if and only if `y` $<_{\texttt{Bed}}$ `x`. The next main object is the genome, which

is a BED file; it is a large text file in the BED format (Neph *et al.*, 2012), the de facto format for this kind of information in the bioinformatics community. A BED file is just a collection of regions, abstracted here as `Vec[Bed]`. The next main object type is the sample, `Sample(bedFile: Vec[Bed], meta: ...)`, which is just a BED file and its associated metadata. The last main object is the sample database, which is just a collection of samples; it is abstracted here as `Vec[Sample]`.

Queries at the level of samples mainly select samples from the sample database to analyze. Queries at the level of BED files mainly extract and process regions of interest. The first kind of queries are basically simplified relational database queries. The second kind of queries are the specialized ones that a relational database cannot handle efficiently. The reason is that these queries invariably have a join predicate which is a conjunction of "genometric" predicates. The GMQL "genometric" predicates in essence are: `DL(n)(y, x)`, meaning the regions overlap or their nearest points are less than `n` bases apart; `DG(n)(y, x)`, meaning the regions do not overlap and their nearest points are more than `n` bases apart; and a few other similar ones. GMQL also implicitly imposes, on genometric predicates, the constraint that `y` and `x` are no further apart than a system-fixed number of bases (e.g., 200,000 bases.) For a reader who is unfamiliar with genomics, "bases," or "bp," is the unit used for describing distance on a genome.

GMQL queries can be easily modeled and efficiently implemented in our Synchrony iterator framework. Let `xs: Vec[Bed]` and `ys: Vec[Bed]` be two BED files sorted in accordance to $<_{\texttt{Bed}}$. Then, `isBefore` is monotonic with respect to $(\texttt{xs}, \texttt{ys})$. Genometric predicates such as `DL(n)` are antimonotonic with respect to `isBefore`. Genometric predicates such as `GL(n)` are not antimonotonic with respect to `isBefore`. As GMQL automatically inserts `DL(200000)` as an additional genometric predicate into a query, a query has at least one antimonotonic genometric predicate.

The implementation of GMQL using Synchrony iterator is described in a companion paper (Perna *et al.*, 2021). Here, we just briefly describe a more complex GMQL query operator, `JOIN(`g_1`, ..., `g_n`; f, h, j)(xss, yss)`. This GMQL query finds all pairs of samples `xs` in `xss` and `ys` in `yss` satisfying the join predicate `j(xs, ys)` on samples. Then for each such pair of samples `xs` and `ys`, for each pair of regions `x` in `xs.bedFile` and `y` in `ys.bedFile` satisfying all the specified genometric predicates g_1`(y, x)`, ..., g_n`(y, x)`, it builds a new region $f(x, y)$; these new regions are put into a new BED file `xys`; finally, a new sample having BED file `xys` and metadata `h(x.meta,y.meta)` is produced.

`JOIN(`g_1`, ..., `g_n`; f, h, j)(xss, yss)` is naturally and efficiently embedded into first-order Scala via comprehension syntax and Synchrony iterator. To wit, it is realized by

```
for (xs <- xss; ys <- yss; if j(x, y))
yield {
  val yi  = new EIterator(ys.bedFile, isBefore, p)
  val xys = for (x <- xs.bedFile; y <- yi.syncedWith(x); if q(y,x))
            yield f(x,y)
  Sample(bedFile = xys, meta = h(xs.meta, ys.meta))
}
```

where `p` is the conjunction of all the antimonotonic predicates among g_1 ..., g_n and `q` is the conjunction of all the remaining predicates among g_1 ..., g_n. In fact, our Synchrony-based GMQL implementation does this decomposition of the list of input genometric predicates into `p` and `q` automatically.

7.3 A stress test

We stress-tested Synchrony iterator by re-implementing GMQL using Synchrony iterator. The GMQL engine (Masseroli *et al.*, 2019) is a state-of-the-art purpose-built system for querying genomic datasets. GMQL is optimized for sample databases containing many samples, with each sample having a large BED file (Neph *et al.*, 2012) containing tens of thousands to hundreds of thousands of genomic regions. GMQL achieves high performance by binning the genome into chunks and comparing different bins concurrently (Gulino *et al.*, 2018).

As the GMQL is based on Scala, we reimplemented it using Synchrony iterator in Scala; this way, the influence of programming language and compiler differences is eliminated. The Synchrony implementation comes with a sequential mode (samples and their BED files are processed in a strictly sequential manner) and a sample-parallel mode (BED files of different samples are processed in parallel but regions in a BED file are processed in a sequential manner.) This reimplementation comprises circa 4,000 lines of Scala codes as counted by cloc; and makes use of some Scala function libraries. In contrast, the original GMQL engine comprises circa 24,000 lines of Scala codes and uses more Scala function libraries and also Spark function libraries. This comparison reveals the merit of Synchrony iterator in enabling complex algorithms to be expressed in a succinct high-level manner.

For benchmarking, we deployed the GMQL engine on a local installation of Apache Spark, which simulates a small cluster on a single multicore machine. We refer to this as the GMQL *command-line interface*, or CLI. The machine is a laptop with 2.6 GHz 6-Core i7, 16 GB 2667 MHz DDR4, 500 GB SSD. Despite the simplicity of our implementation, it significantly outperforms GMQL CLI on essentially all test queries and on the full range of dataset sizes and equals GMQL CLI on the largest-size datasets. This is a strong testimony to Synchrony iterator as an elegant idea for expressing efficient synchronized iterations on multiple collections in a succinct and easy-to-understand manner. The implementation and detailed evaluation are presented in a companion paper (Perna *et al.*, 2021). The implementation is available at `https://www.comp.nus.edu.sg/~wongls/projects/synchrony`.

We present below some comparison results on a simple region `MAP` query. The GMQL `MAP` query takes two sample databases `xss` and `yss` and produces for each pair of BED files `xs.bedFile` in `xss` and `ys.bedFile` in `yss`, and each region `x` in `xs.bedFile`, the number of regions in `ys.bedFile` that it overlaps with. GMQL executes its `MAP` operator in a four-level deeply nested loop, in a brute-force parallel manner; i.e., all BED file pairs are analyzed in parallel. For each BED file pair, the BED files are chopped into bins; the bins are paired; and all bin pairs are analyzed in parallel. Ignoring parallelism, the complexity is $O(n^2m^2)$ assuming both `xss` and `yss` contain n BED files and each BED file contains $m >> n$ regions. The Synchrony iterator version uses a two-level nested loop to pair up the BED files, but each pair of BED files is analyzed using a Synchrony iterator:

```
for (xs <- xss; ys <- yss)
yield {
  val yi = new EIterator(ys.bedFile, isBefore, DL(0))
  for (x <- xs.bedFile; r = yi.syncedWith(x))
  yield (x, r.length)
}
```

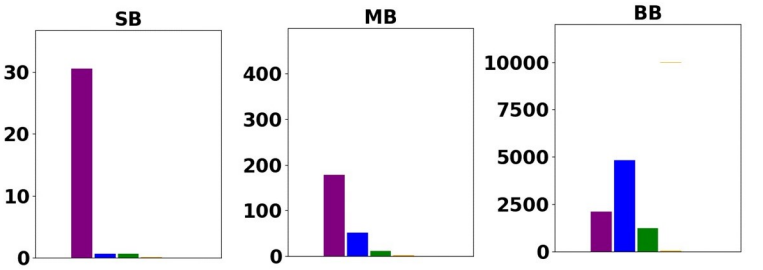

Fig. 12. Performance of GMQL CLI and Synchrony emulation on simple region `MAP`. Time in seconds, average of 30 runs for SB and MB, and 5 runs for BB. *Purple:* GMQL CLI. *Blue:* Sequential Synchrony emulation. *Green:* Sample-parallel Synchrony emulation.

The sample-parallel version runs the two-level nested loop in parallel but the Synchrony iterator sequentially. The sequential version does everything sequentially. Ignoring parallelism, the complexity is $O((2k+1)mn^2)$ where k is a small number corresponding to the maximum number of overlaps a region can have with other regions.

For this paper, the three versions are run on three input settings (SB, MB, BB) containing varying number of BED files, where each BED file has 100,000 regions. The setting SB means both `xss` and `yss` contain exactly one BED file; thus, there is exactly one BED file pair to analyze. The setting MB means both `xss` and `yss` contain exactly ten BED files; thus, there are 100 BED file pairs to analyze. The setting BB means both `xss` and `yss` contain exactly one hundred BED files; thus, there are 10,000 BED file pairs to analyze. Roughly `xss` and `yss` are each of size circa 10MB, 96MB, and 696MB on disk in settings SB, MB, and BB. The timings are shown in Figure 12. It is clear that the two Synchrony iterator versions are far more efficient than GMQL CLI. Only in the BB setting, GMQL CLI is able to beat the strict sequential Synchrony iterator. But GMQL CLI's brute-force parallelism is still no match to sample-parallel Synchrony iterator for these and other settings considered.

7.4 Stream queries

As a last use-case, we model timestamped data streams. Two kinds of objects are considered for this purpose. The first kind is called observations. An observation `x: Obs(at: Int, id: ...)` has a timestamp `x.at`, which is the time the observation is obtained, and has some other pieces of information that are irrelevant for our purpose. All timestamps are the number of nanoseconds that have elapsed since a fixed reference time point. The second kind is called observation streams. An observation stream is just a collection of observations, `xs: Vec[Obs]`.

Observations are intrinsically ordered by their timestamps. Thus, it is natural to define the following as the `isBefore` predicate on observation streams:

```
bf(y, x) = y.at < x.at
```

and it is also natural to assume that observation streams are sorted by timestamps by default. A variety of `canSee` predicates can be easily defined, such as:

```
notAfter(y, x) = ! (y.at > x.at)
```

```
within(n)(y, x) = abs(y.at - x.at) <= n
```

For convenience, let `clk` be a stream of observations representing regular clock ticks at intervals of 1 milliseconds. Also, let `xss`, `yss`, and `zss` be several streams of observations. Then a variety of observation processing can be easily and efficiently expressed. Just for practice, to see how it looks, the suggested syntax for abstracting away Synchrony iterator from Section 6.4 is used here; the programs below are not legitimate Scala.

- `cartesian(f)(clk, xss, yss, zss)` groups observations in `xss`, `yss`, and `zss` into 1 millisecond time-synchronized blocks; applies `f` to each block to generate a new observation stream.

```
cartesian(f)(clk, xss, yss, zss)
= { val cs= (u:Obs, v:Obs) => within(1000)(u,v) && notAfter(u,v)
    for ((c, xs, ys, zs) <- clk syncWith(xss, bf, cs)
                                syncWith(yss, bf, cs)
                                syncWith(zss, bf, cs)
    ) yield f(c, xs, ys, zs) }
```

- `mostRecent(f)(clk, xss, yss, zss)`, applies `f` to the last observation in `xss`, `yss`, and `zss` within each 1 millisecond block. Skips a block if any stream contains no observation in that block of time.

```
mostRecent(f)(clk, xss, yss, zss)
= { val cs= (u:Obs, v:Obs) => within(1000)(u,v) && notAfter(u,v)
    for ((c, xs, ys, zs) <- clk syncWith(xss, bf, cs)
                                syncWith(yss, bf, cs)
                                syncWith(zss, bf, cs);
          if ! (xs.isEmpty || ys.isEmpty || zs.isEmpty)
    ) yield f(c, xs.last, ys.last, zs.last) }
```

- `affineMostRecent(f)(clk, xss, yss, zss)`, applies `f` to the last observation in `xss`, `yss`, and `zss` within each 1 millisecond block. If a block has a stream which contains no observation in this block of time, keep observations in this block and consider them with the next block.

```
affineMostRecent(f)(clk, xss, yss, zss)
= { val cs = (u:Obs, v:Obs) => within(2000)(u,v) && notAfter(u,v)
    def nd(us: Vec[Obs], t: Int) = us.filter(_.at <= t).isEmpty
    for ((c, xs, ys, zs) <- clk syncWith(xss, bf, cs)
                                syncWith(yss, bf, cs)
                                syncWith(zss, bf, cs);
         if !(xs.isEmpty || ys.isEmpty || zs.isEmpty);
         (lx, ly, lz, oc) = (xs.last, ys.last, zs.last, c.at - 999);
         if (lx.at > oc && ly > oc && lz > oc) ||
            ((lx.at > oc || ly > oc || lz > oc) &&
             (nd(xs, oc) || nd(ys, oc) || nd(zs, oc)))
    ) yield f(c, lx, ly, lz) }
```

As can be seen, a variety of temporal stream processing and synchronization operators, akin to those in Bracevac *et al.* (2018), can be implemented in comprehension syntax using Synchrony iterator. Notably, provided there are not too many events within each 1 millisecond block and f has at most linear time complexity, all of these examples have linear time complexity. If the observation type has a structure that carries more information (e.g., length of observation, if observation extends over a period of time), an even richer variety of antimonotonic predicates can be defined and used in specifying stream synchronization.

To some extent, this use-case illustrates that Synchrony iterator is not restricted to database query processing. Rather, it is capturing and generalizing common patterns and forms of synchronized iterations, such as those found in database query processing and in stream event processing.

8 Other possibilities

8.1 Grouping

It is instructive to look at the function `groups` in the top half of Figure 13. It was suggested by a reviewer as an approach to efficient implementation of relational joins. The idea is based on grouping. This suggestion inspired us to introduce the Synchrony generator `syncGenGrp`, which we did not describe in the initial draft of this paper mainly because it returns a nested collection and thus, strictly speaking, does not meet the first-order restriction requirement.

The reviewer probably had in mind a function like `syncGenGrp` and provided the function `groups` in Figure 13 as the implementation. However, this only works correctly when the join predicate `cs` is a conjunction of equality tests, i.e. an equijoin. Here is an example to show that it does not correctly implement a join in general. Let us regard `xs: Vec[Event]` and `ys: Vec[Event]` as lists of line segments sorted by (`start`, `end`). Consider the following line segments.

```
a = Event(start = 10, end = 70, id = "a")
b = Event(start = 20, end = 30, id = "b")
c = Event(start = 40, end = 80, id = "c")
d = Event(start = 60, end = 90, id = "d")
```

Let `isBefore` and `overlap` be as defined in Figure 1. Let `xs` be a singleton containing the line segment `d` and `ys` comprises the line segments `a`, `b` and `c` in this order. Then `ov1(xs, ys)` evaluates to exactly the two pairs `(d, a)` and `(d, c)`. In agreement with `ov1(xs, ys)`, `syncGenGrp(isBefore, overlap)(xs, ys)` evaluates to the singleton `(d, Vec(a, c))`. Whereas, `groups(isBefore, overlap)(xs, ys)` incorrectly evaluates to an empty collection.

Perhaps instead of `val yt = ys.dropWhile(y => bf(y, x))`, the reviewer meant `val yt = ys.dropWhile(y => bf(y, x) && !cs(y, x))`. This revised `groups(bf, cs)(xs, ys)` works correctly when `bf` is monotonic with respect to (`xs`, `ys`) and `cs` is reflexive and convex with respect to `bf`. It does not work as expected when `cs` is antimonotonic but not convex. We mentioned earlier that predicates which are reflexive and convex are also antimonotonic, and that the converse is not true. The `overlap` predicate on events is such

```
def groups[A,B]
  (bf: (B,A) => Boolean, cs: (B,A) => Boolean)
  (xs: Vec[A], ys: Vec[B])
: Vec[(A,Vec[B])] = {
  def step(acc: (Vec[(A,Vec[B])], Vec[B]), x: A)
  : (Vec[(A, Vec[B])], Vec[B]) = {
    val (xzss, ys) = acc
    // this works only for equijoin cs:
    val yt = ys.dropWhile(y => bf(y, x))
    // this works for convex cs:
    // val yt = ys.dropWhile(y => bf(y, x) && ! cs(y, x))
    val zs = yt.takeWhile(y => cs(y, x))
    (xzss :+ (x, zs), yt)
  }
  val e: (Vec[(A,Vec[B])], Vec[B]) = (Vec(), ys)
  val (xzss, _) = xs.foldLeft(e)(step _)
  return xzss
}


def groups2[A,B]
  (bf: (B,A) => Boolean, cs: (B,A) => Boolean)
  (xs: Vec[A], ys: Vec[B])
: Vec[(A,Vec[B])] = {
  // Requires: bf monotonic wrt (xs, ys); cs antimonotonic wrt bf.
  val step = (acc: (Vec[(A,Vec[B])], Vec[B]), x: A) => {
    val (xzss, ys) = acc
    val maybes = ys.takeWhile(y => bf(y, x) || cs(y, x))
    val yes = maybes.filter(y => cs(y, x))
    val nos = ys.dropWhile(y => bf(y, x) || cs(y, x))
    (xzss :+ (x, yes), yes ++: nos)
  }
  val e: (Vec[(A,Vec[B])], Vec[B]) = (Vec(), ys)
  val (xzss, _) = xs.foldLeft(e)(step)
  return xzss
}
```

Fig. 13. Alternative attempts to define `syncGenGrp`. The function `groups` is only correct when `cs` is an equijoin predicate. The function `groups2` is equivalent to `syncGenGrp` and has comparable efficiency.

an example; it is antimonotonic but not convex. This can be seen using the line segments given above: `overlap(a, d)` and `overlap(c, d)` but not `overlap(b, d)`. Indeed, this revised `groups` function returns the singleton `(d, Vec(a))`, which is still incorrect.

In order to get the correct semantics as `syncGenGrp`, the definition of `groups` must be modified to account for antimonotonicity. This can be done as in `groups2`, depicted in the bottom half of Figure 13. In `groups2`, for each `x` in `xs`, `step` iterates on the current copy of `ys`, to divide it using `takeWhile` and `dropWhile`. The function `dropWhile` stops the iteration on `ys` as soon as an item `y` in `ys` is encountered such that both `bf(y, x)` and `cs(y, x)` are false, and yields the remainder `nos`. This early stopping is correct due to Antimonotonicity Condition 2. The function `takeWhile` copies items on `ys` until an item `y` in `ys` is encountered such that both `bf(y, x)` and `cs(y, x)` are false, obtaining the prefix `maybes`. Those items in `maybes` that can see `x` are extracted into `yes` and returned as the result for this `x`. The function `step` also updates `ys` to `yes ++: nos`, "rewinding" it and setting it up for the next item in `xs`. Thus, for the next `x`, the iteration on `ys` effectively skips over all the items in `ys` that are before it and cannot see it. This skipping is correct due to Antimonotonicity Condition 1.

`Group2` and `syncGenGrp` can be shown to define the same function and have similar time complexity. The concepts of monotonicity and antimonotonicity seems fundamental to achieving efficient synchronized iteration. In particular, `groups2` needs both of these to ensure correctness. Their use in `groups2` has further clarified how these concepts interact to synchronize iteration. Specifically, synchronized iteration on two collections, which are already sorted in a comparable ordering, is characterized by knowing precisely when to start, stop, and "rewind." Hence, by explicitly parameterizing its iteration control on monotonicity and antimonotonicity, Synchrony fold can perhaps be regarded as a programming construct that characterizes efficient synchronized iteration on ordered collections.

8.2 Indexed tables

A reviewer also introduced us to the recent works of Gibbons (2016) and Gibbons *et al.* (2018), which investigate programming language embedding of joins that avoid naive evaluation strategies. These works provide an elegant mathematical foundation for indexing and grouping, leading to efficient implementation of equijoins. Underlying the theoretical perspective of these works is the use of indexed tables.

Indexed tables are part of the repertoire of collection-type libraries of modern programming languages. For example, the collection-type libraries of Scala provide a `groupBy` method. For a collection `ys: Vec[A]`, and an indexing function `f: A => K`, `ys.groupBy(f)` builds and returns an indexed table `ms: Map[K,Vec[A]]`, where `ms(ky)` is the precomputed result of `for (y <- ys; if f(y) == ky) yield y`. Assuming `f` is a function of $O(1)$ time complexity, the indexed table is constructed in $O(|\texttt{ys}|)$ time, and accessing `ms(f(y))` takes $O(1)$ time.

This `groupBy` function is useful for implementing efficient equijoin by a user-programmer, if we disregard the fact that it returns a nested collection and is thus not first order. For a direct example, assuming `f` and `g` are constant-time functions, the equijoin `for (x <- xs; y <- ys; if g(x) == f(y)) yield (x, y)` can be computed in $O(|\texttt{xs}| + |\texttt{ys}|)$ time using an indexed table as `{ val ms = ys.groupBy(f); for (x <- xs; y <- ms(g(x)) yield (x,y) }`. This corresponds to an index-seek join strategy, which a database system usually uses when it expects there are not many `y` that matches any `x` at all. As another example, Gibbons (2016) describes an interesting perspective where collections are viewed as indexed tables, and derives a zip-parallel comprehension for joining them. This strategy correponds to the index-scan join strategy, which a database system uses when it expects most `x` matches at least one `y` and vice versa.

Using an indexed-table approach to implement an equijoin has a key advantage that the input collections do not need to be sorted to begin with. Provided all indexed tables which are needed can fit into memory, implementing an equijoin using an indexed-table approach is superior to using Synchrony iterator. If one or more of the input collections are unsorted or are in some unsuitable orderings, these inputs have to be sorted before Synchrony iterator can be used to process them; this can be a significant overhead when the input collections which require sorting are large.

On the other hand, there are several limitations with indexed tables, especially when we are not operating on an actual database system. Firstly, this means the indexed table

is an in-memory structure; so, it is not suitable for very large collections.[10] Secondly, the indexed table has to be completely constructed before it is used; so, it is not suitable for data streams. Lastly, and crucially, as an indexed table relies on exact equality to directly retrieve entries, it can easily implement efficient equijoin but it cannot implement non-equijoin such as band join and interval join.

In contrast, Synchrony iterator does not suffer these limitations. In other words, Synchrony iterator is a more general and more uniform approach for realizing efficient equijoin and a large class of non-equijoin. Synchrony iterator is thus justifiably appealing.

9 Concluding remarks

Modern programming languages typically provide some form of comprehension syntax for manipulating collection types. In this regard, comprehension syntax does not add extensional expressive power, but it makes programs much more readable (Trinder, 1991; Buneman *et al.*, 1994). Comprehensions typically correspond to nested loops. So, it is difficult to use comprehension syntax to express efficient algorithms for, e.g., database joins. This has partly motivated developments that introduced alternative binding semantics for comprehension syntax, so that some comprehensions are not compiled into nested loops. For example, parallel and grouping comprehension were introduced to enable implementation of efficient database queries in the style of comprehension syntax (Wadler & Peyton Jones, 2007; Gibbons, 2016; Gibbons *et al.*, 2018). Nonetheless, it has not been formally demonstrated that efficient algorithms for, e.g., equijoin cannot be implemented without making such refinements to comprehension syntax.

The first contribution of this paper is to highlight, in a precise sense, comprehension syntax suffers a limited-mixing handicap. In particular, this formally confirms that efficient algorithms for low-selectivity database joins—and this includes equijoin—cannot be implemented using comprehension syntax in the first-order setting (i.e., first-order restricted Scala in the context of this paper.) This justifies, from the intensional expressive power point of view, that these interesting works are necessary.

Although there is no efficient implementation for low-selectivity database joins in the first-order setting, they are nonetheless expressible as functions in the first-order setting. Therefore, the gap is purely in the intensional expressive power of comprehension syntax. So, we considered whether any function commonly provided in the collection-type libraries of modern programming languages is able to fix this gap. The limited-mixing handicap of comprehension syntax in the first-order setting remains even after adding any one of `foldLeft`, `takeWhile`, `dropWhile`, and `zip`; and most common functions in these libraries are derivatives of `foldLeft`.

[10] Even when we are operating on an actual database system, this fits-into-memory issue can be a problem in a situation where the database system has to—as is often the case—process many queries concurrently. While hash tables needed by a query may fit into memory, this may prevent hash tables needed by other queries to fit, thereby affecting the overall performance of the system. This was a reason that even though the hash join (Silberschatz *et al.*, 2016), which is based on dynamically constructing a hash table, has been known and implemented in database systems a long time ago, its use was discouraged (e.g., hash join was routinely disabled in Oracle 11g systems) until recent times when systems with very large memory have become common.

The second contribution of this paper is to identify and propose a candidate library function which fills this gap. We noticed that, apart from `zip`, the notion of general synchronized iteration on multiple collections is conspicuously absent from current collection-type libraries. Hence, to kill two birds with one stone, we looked for a function that encapsulates a common pattern of general synchronized iteration on multiple collection. Arguably, `foldLeft` is the most powerful function in collection-type libraries of modern programming languages, as most other commonly found functions in these libraries are extensionally expressible using `foldLeft`. So, as an upperbound, we identified Synchrony fold, which is a novel synchronized iteration function that expresses the same functions as `foldLeft` in the first-order setting and yet expresses more algorithms, including efficient low-selectivity database joins. Furthermore, just as a simple restriction can be imposed on `foldLeft` to cut its extensional expressive power to precisely match comprehension syntax, a similar restriction can be imposed on Synchrony fold to cut its extensional expressive power to precisely match comprehension syntax. This restricted form is Synchrony generator. Synchrony generator expresses exactly the same functions as comprehension syntax in the first-order setting; but it expresses a richer repertoire of algorithms, including efficient low-selectivity database joins. Hence, Synchrony generator is a conservative extension of comprehension syntax that precisely fills its intensional expressiveness gap.

Synchrony generator is nonetheless not well dovetailed with comprehension syntax in the first-order setting. In particular, synchronized iteration over multiple ordered collections simultaneously apparently can only be expressed using Synchrony generator in an aesthetically clumsy manner in the first-order setting. When a function $\texttt{zip}_n$ for simultaneously zipping n collections is available, efficient synchronized iteration over n collections can be succinctly and elegantly expressed using Synchrony generator and this function. However, $\texttt{zip}_n$ is outside the first-order setting. Moreover, this approach carries overheads of n extra scans of at least one dataset. Another limitaton of this approach is that it is not user-programmer friendly: A zoo of $\texttt{zip}_3$, $\texttt{zip}_4$, etc. have to be provided, as a single `zip` function for zipping an arbitrary number of collections of different types cannot be assigned a valid type in strongly typed programming languages.

The third contribution of this paper is Synchrony iterator. We found that Synchrony generator is algorithmically equivalent to iterating on the items in a first collection, and invoking Synchrony iterator on each of these items to efficiently return matching items in a second collection. Synchrony iterator thus smoothly dovetails with comprehension syntax. More importantly, it enables efficient synchronized iteration on multiple collections to be simply expressed in comprehension syntax in a first-order setting and without the n extra-scan overheads.

Synchrony fold, Synchrony generator, and Synchrony iterator can be regarded as capturing an intuitive pattern of efficient synchronized iteration on ordered collections. They suggest that efficient synchronized iteration on ordered collections are characterized by a monotonic `isBefore` predicate that relates the orderings of the input collections, and an antimonotonic `canSee` predicate that identifies matching pairs to act on. The antimonotonicity conditions on `canSee` further informs that the efficiency of the synchronization arises from exploiting "right-sided convexity" of the matching items. Indeed, together, these predicates make explicit where to start, stop, and rewind an iteration on two collections, thereby achieving efficient synchronization.

The fourth contribution of this paper is the revelation that efficient synchronized iteration on ordered collections is captured by such a pattern which is characterized by the monotonic `isBefore` and antimonotonic `canSee` predicates. A corollary of this fourth contribution is the result that Synchrony generator (and thus Synchrony iterator) is a natural generalization of the merge join algorithm (Blasgen & Eswaran, 1977; Mishra & Eich, 1992) widely used in database systems for decades for realizing efficient equijoins. With a simple modification implied by Synchrony generator, the modified merge join algorithm can work as long as the join predicate is antimonotonic with respect to the sort order of the relations being joined.

Lastly, we briefly described using Synchrony iterator to re-implement GMQL (Masseroli *et al.*, 2019), which is a state-of-the-art query system for large genomic datasets. The Synchrony-based re-implementation is more efficient than GMQL, and is also six-fold shorter in terms of number of lines of codes, thereby validating the theory and design of Synchrony fold and Synchrony iterator.

This paper primarily illustrates Synchrony fold, Synchrony generator, and Synchrony iterator using examples based on low-selectivity database joins. Nonetheless, Section 7.4 briefly showcases using Synchrony iterator to specify event stream processing operators. This suggests Synchrony fold and Synchrony iterator capture patterns of efficient synchronized iteration, showing that they can be parameterized by a pair of monotonic `isBefore` and antimonotonic `canSee` predicates. However, our notion of synchronized iteration, as encapsulated by Synchrony fold, generator, and iterator, is quite constrained. It maybe a worthwhile future work to understand what interesting yet common patterns of efficient synchronized iteration are not encapsulated by Synchrony fold and Synchrony iterator.

Conflicts of Interest

None

Acknowledgements

Stefano Ceri invited us to the *GeCo Workshop on Challenges in Data-Driven Genomic Computing*, held in Como, Italy, in March 2019. This work evolved from the talk given by LW at the workshop and the ensuing interesting discussions with VT and SP. We thank Stefano for his invitation and surreptitious seeding of this work.

Jeremy Gibbons and the reviewers provided very useful suggestions on this paper. They also brought many relevant works and ideas to our attention. Their comments greatly enriched our perspective in this work. We thank them for their invaluable contribution in helping us improve this work.

This work was supported by National Research Foundation, Singapore, under its Synthetic Biology Research and Development Programme (Award No: SBP-P3); and by Ministry of Education, Singapore, Academic Research Fund Tier-1 (Award No: MOE T1 251RES1725). In addition, VT was supported in part by a Kwan Im Thong Hood Cho Temple Visiting Professorship, and LW was supported in part by a Kwan Im Thong Hood

Cho Temple Chair Professorship. Any opinions, findings, and recommendations expressed herein are those of the authors, and do not reflect the views of these grantors.

References

Abiteboul, S. and Vianu, V. (1991) Generic computation and its complexity. *Proceedings of 23rd ACM Symposium on the Theory of Computing* pp. 209–219.

Biskup, J., Paredaens, J., Schwentick, T. and den Bussche, J. V. (2004) Solving equations in the relational algebra. *SIAM Journal on Computing* **33**(5):1052–1066.

Blasgen, M. and Eswaran, K. (1977) Storage and access in relational databases. *IBM Systems Journal* **16**(4):363–377.

Bracevac, O., Amin, N., Salvaneschi, G., Erdweg, S., Eugster, P. and Mezini, M. (2018) Versatile event correlation with algebraic effects. *Proceedings of ACM on Programming Languages* **2**(ICFP):67.

Buneman, P., Libkin, L., Suciu, D., Tannen, V. and Wong, L. (1994) Comprehension syntax. *SIGMOD Record* **23**(1):87–96.

Buneman, P., Naqvi, S., Tannen, V. and Wong, L. (1995) Principles of programming with complex objects and collection types. *Theoretical Computer Science* **149**(1):3–48.

Colson, L. (1991) About primitive recursive algorithms. *Theoretical Computer Science* **83**:57–69.

DeWitt, D. J., Naughton, J. F., Schneider, D. A. (1991) An evaluation of non-equijoin algorithms. *Proceedings of 17th International Conference on Very Large Data Bases* pp. 443–452.

Dignoes, A., Boehlen, M. H., Gamper, J., Jensen, C. S. and Moser, P. (2021) Leveraging range joins for the computation of overlap joins. *The VLDB Journal*, https://doi.org/10.1007/s00778-021-00692-3.

Felleisen, M. (1991) On the expressive power of programming languages. *Science of Computer Programming* **17**:35–75.

Fortune, S., Leivant, D. and O'Donnell, M. (1983) The expressiveness of simple and second-order type structures. *Journal of the ACM* **30**(1):151–185.

Gaifman, H. (1982) On local and non-local properties. *Proceedings of the Herbrand Symposium, Logic Colloquium '81* pp. 105–135. North Holland.

Gibbons, J. (2016) Comprehending ringads. Lindley, S., McBride, C., Trinder, P. and Sannella, D. (eds), *A List of Successes That Can Change the World* pp. 132–151.

Gibbons, J., Henglein, F., Hinze, R. and Wu, N. (2018) Relational algebra by way of adjunctions. *Proceedings of the ACM on Programming Languages* **2**(ICFP):86.

Gulino, A., Kaitoua, A. and Ceri, S. (2018) Optimal binning for genomics. *IEEE Transactions on Computers* **68**(1):125–138.

Henglein, F. and Larsen, K. F. (2010) Generic multiset programming with discrimination-based joins and symbolic Cartesian products. *Higher-Order and Symbolic Computation* **23**(3):337–370.

Hunt, A. and Thomas, D. (2000) *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley.

Knuth, D. E. (1973) *The Art of Computer Programming: Sorting and Searching*. Addison Wesley.

Libkin, L. and Wong, L. (1994) Aggregate functions, conservative extension, and linear orders. Beeri, C., Ohori, A. and Shasha, D. E. (eds), *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993* pp. 282–294. Springer-Verlag. See also UPenn Technical Report MS-CIS-93-36.

Libkin, L. and Wong, L. (1997) Query languages for bags and aggregate functions. *Journal of Computer and System Sciences* **55**(2):241–272.

Lindley, S., Wadler, P. and Yallop, J. (2011) Idioms are oblivious, arrows are meticulous, monads and promiscuous. *Eletronic Notes in Theoretical Computer Science* **229**(5):97–117.

Marlow, S., Peyton-Jones, S., Kmett, E. and Mokhov, A. (2016) Desugaring Haskell's do-notation into applicative operations. *ACM SIGPLAN Notices* **51**(12):92–104.

Masseroli, M., Canakoglu, A., Pinoli, P., Kaitoua, A., *et al.* . (2019) Processing of big heterogeneous genomic datasets for tertiary analysis of next generation sequencing data. *Bioinformatics* **35**(5):729–736.

Mishra, P. and Eich, M. H. (1992) Join processing in relational databases. *ACM Computing Surveys* **24**(1):63–113.

Neph, S., Kuehn, M. S., Reynolds, A. P., Haugen, E., Thurman, R. E., Johnson, A. K., Rynes, E., Maurano, M. T., Vierstra, J., Thomas, S., Sandstorm, R., Humbert, R. and Stamatoyannopoulos, J. A. (2012) BEDOPS: High-performance genomic feature operations. *Bioinformatics* **28**(4):1919–1920.

Odersky, M., Spoon, L. and Venners, B. (2019) *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Inc.

Olteanu, D. and Schleich, M. (2016) Factorized databases. *ACM SIGMOD Record* **45**(2):5–16.

Perna, S., Pinoli, P., Tannen, V., Ceri, S. and Wong, L. (2021) *Synchronized iteration for genomic data processing*. Manuscript available from https://www.comp.nus.edu.sg/~wongls/projects/synchrony/synchrony-gmql-v12.pdf.

Piatov, D., Helmer, S. and Dignoes, A. (2016) An interval join optimized for modern hardware. *Proceedings of 32nd IEEE International Conference on Data Engineering* pp. 1098–1109.

Schmidt, D. A. (1986) *Denotational Semantics: A Methodology For Language Development*. Allyn and Bacon.

Sebesta, R. W. (2010) *Concepts of Programming Languages*. Addison-Wesley.

Silberschatz, A., Korth, H. F. and Sudarshan, S. (2016) *Database System Concepts*. 7th edn. McGraw-Hill.

Suciu, D. and Paredaens, J. (1997) The complexity of the evaluation of complex algebra expressions. *Journal of Computer and Systems Sciences* **55**(2):322–343.

Suciu, D. and Wong, L. (1995) On two forms of structural recursion. *LNCS 893: Proceedings of 5th International Conference on Database Theory* pp. 111–124. Springer-Verlag.

Trinder, P. W. (1991) Comprehensions, a query notation for DBPLs. *Proceedings of 3rd International Workshop on Database Programming Languages, Nahplion, Greece* pp. 49–62. Morgan Kaufmann.

Van den Bussche, J. (2001) Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science* **254**(1–2):363–377.

Wadler, P. (1990) Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science* **73**:231–248.

Wadler, P. and Peyton Jones, S. (2007) Comprehensive comprehension: Comprehensions with 'order by' and 'group by'. *Haskell '07: Proceedings of ACM SIGPLAN Workshop on Haskell* pp. 61–72.

Wong, L. (1996) Normal forms and conservative extension properties for query languages over collection types. *Journal of Computer and System Sciences* **52**(3):495–505.

Wong, L. (2013) A dichotomy in the intensional expressive power of nested relational calculi augmented with aggregate functions and a powerset operator. *Proceedings of 32nd ACM Symposium on Principles of Database Systems* pp. 285–295.

Wong, L. (2021) *Addressing an intensional expressiveness gap of comprehension syntax*. Manuscript available from https://www.comp.nus.edu.sg/~wongls/projects/synchrony/v5-wls-natural2021.pdf