Iterating on multiple collections in synchrony

Journal of Functional Programming 32:e9, 2022

Stefano Perna, Val Tannen, **Limsoon Wong**

Motivating example

When xs and ys are sorted according to isBefore,

ov1(xs, ys) = ov2(xs, ys)

ov1(xs,ys) has complexity O(|xs|·|ys|)

ov2(xs,ys) has complexity O(|xs| + k |ys|), where each event in ys overlaps fewer than k events in xs

Can we get the simplicity of ov1 at the efficiency of ov2?

```scala
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)
}
```

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

```scala
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())
}
```

Intensional expressiveness gap

ov1 is easily expressible using only comprehension syntax

No obvious efficient implementation w/o using more advanced programming language features and/or library functions

Many other functions suffer the same plight …

{ (x,y) | x, y $\in$ taxpayers, x earns less but pays more tax than y }

{ (x,y) | x, y $\in$ mobile phones, x’s price is similar to y’s price }

Limited mixing lemma

Comprehension in restricted 1^{st} order setting

Let e(X) be an expression in $NRC_1(<)$ and $e[C/X] \Downarrow C'$. Suppose e(X) has at most linear-time complexity wrt size of X. Then for each (u,v) in gaifman(C’), either

(u,v) in gaifman(C), or

u in $atom^0(C)$ and v in $atom^1(C)$, or

u in $atom^1(C)$ and v in $atom^0(C)$

Similar limited mixing lemmas can be proved for

NRC_1(takewhile, dropwhile, sort,<)

NRC_1(foldleft, sort,<)

NRC_1(zip, sort,<)

Types in $\mathcal{NRC}_1$

$$t ::= b \mid b_1 \times \cdots \times b_n$$
$$s ::= t \mid \{t\} \mid s_1 \times \cdots \times s_n$$
where b’s are base types.

Expressions in $\mathcal{NRC}_1$

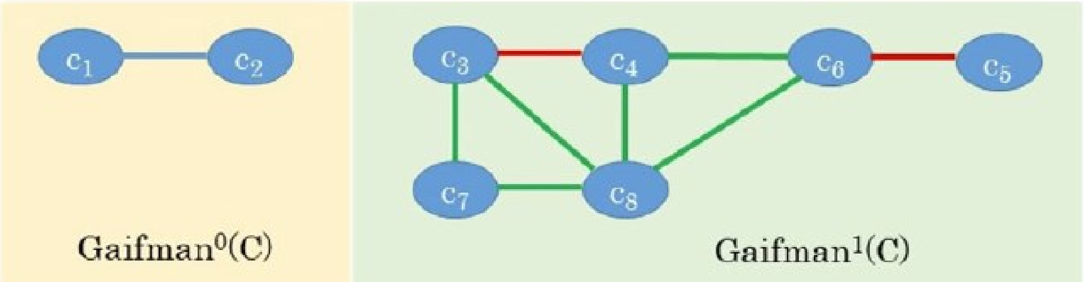
$$\frac{}{C^s : s} \qquad \frac{}{x^s : s} \qquad \frac{e_1 : b_1 \quad \ldots \quad e_n : b_n}{(e_1, \ldots, e_n) : b_1 \times \cdots \times b_n} \qquad \frac{e : b_1 \times \cdots \times b_n}{e.\pi_i : b_i}\, 1 \le i \le n$$

$$\frac{}{\{\}^t : \{t\}} \qquad \frac{e : t}{\{e\} : \{t\}} \qquad \frac{e_1 : \{t\} \quad e_2 : \{t\}}{e_1 \cup e_2 : \{t\}} \qquad \frac{e_1 : \{t_1\} \quad e_2 : \{t_2\}}{\bigcup\{e_1 \mid x^{t_2} \in e_2\} : \{t_1\}}$$

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

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

Gaifman(C), for C = $(c_1, c_2, \{(c_3, c_4), (c_5, c_6)\}, \{(c_3, c_7, c_8), (c_4, c_8, c_6)\})$



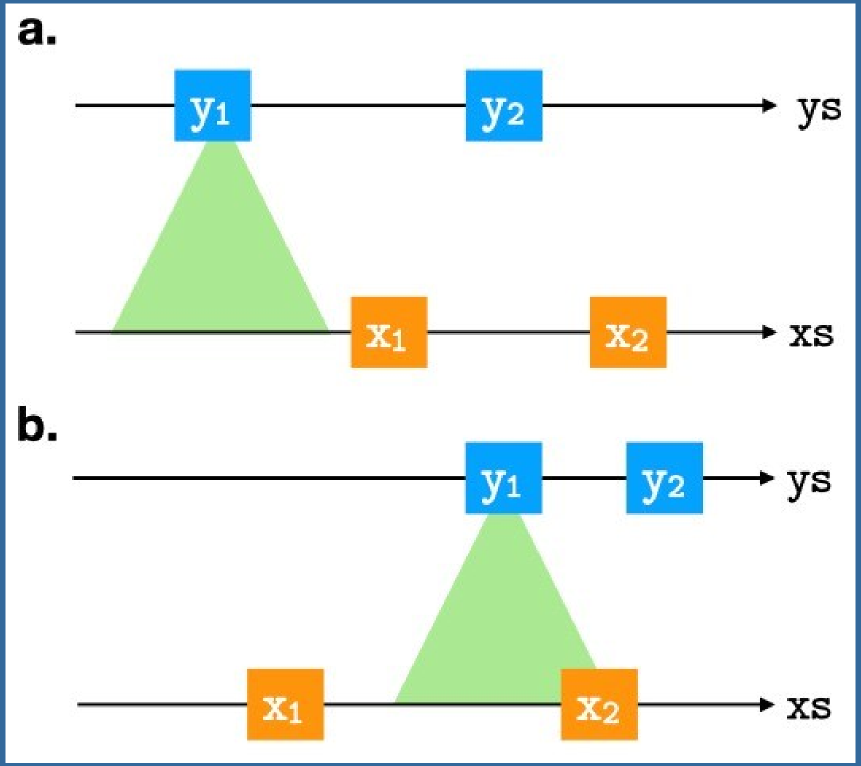
$atom^0(C) = \{ c_1, c_2 \}$; $atom^1(C) = \{ c_3, c_4, c_5, c_6, c_7, c_8 \}$

Intensional expressiveness gap is “real”

What new library function or programming construct *precisely* fills the gap?

I.e., how to allow the “missing” efficient *algorithms* to be expressed w/o changing the class of *functions* that can be expressed

Monotonicity & antimonotonicity



Monotonicity of** bf **wrt (xs, ys)

If (x « x’ | xs), then ∀y in ys: bf(y, x) implies bf(y, x’)

If (y’ « y | ys), then ∀x in xs: bf(y, x) implies bf(y’, x)

***Antimonotonicity of** cs **wrt** bf*

If (x « x’ | xs), then ∀y in ys: bf(y, x) & ! cs(y, x) implies ! cs(y, x’)

If (y « y’ | xs), then ∀x in xs: ! bf(y, x) & ! cs(y, x) implies ! cs(y’, x)

Synchrony generator, capturing a programming pattern for efficient synchronized iteration on two collections

When bf/isBefore is monotonic wrt (xs, ys) and cs/overlap is antimonotonic wrt bf :

ov1(xs, ys) = ov4(xs, ys)

ov1(xs,ys) has complexity O(|xs| ·|ys|)

ov2(xs,ys) has complexity O(|xs| + k|ys|), where each event in ys overlaps fewer than k events in xs

```
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())
}
```

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.

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.

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

```
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.
  for (x <- xs, (_, Y) <- syncGenGrp(isBefore, overlap)(xs, ys), y <- Y) yield (x, y)
}
```

syncGenGrp is a conservative extension

The functions definable in $NRC_1(<)$ and $NRC_1(<,syncGenGrp)$ are exactly the same

However, more efficient algorithms for some functions (e.g., low-selectivity joins) are definable in the latter

Thus, syncGenGrp fills the intensional expressive power gap of comprehension syntax in a “1st-order restricted setting”

A zoo of relational joins

Defined based on syntactic restrictions on join predicates

Implemented by different algos for efficiency

type	form	usual implementation	properties
equijoin	x.a = y.b	hash join, merge join	convex, reflexive
single inequality	x.a ≤ y.b	merge join	Convex, reflexive
range join	x.a – e ≤ y.b ≤ x.a + e	range join	Convex, reflexive
band join	x.a ≤ y.b ≤ x.c	band join	Convex, reflexive
interval join	x.a ≤ y.b && y.c ≤ x.d where x.a ≤ x.d and y.c ≤ y.b	Union of two band joins, interval joins for special data types	Non-convex, antimonotonic

Convexity ⇒ antimonotonicity

∴ syncGenGrp implements them simply and efficiently, viz. Synchrony join

syncGenGrp generalizes relational merge join from equijoin to antimonotonic predicates

```
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
}
```

groups = merge join algo, implements relational join when cs is an equijoin predicate

{ (x, y) | x <- xs, (_,Y) <- groups(bf, cs)(xs, ys), y <- Y }

= join { (x, y) | x <- xs, y <- ys, cs(y, x) }

```
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
}
```

groups2 = syncGenGrp extensionally & intensionally

groups2 = a novel “synchrony” join algo, implements relational join when cs is an antimonotonic predicate

{ (x, y) | x <- xs, (_,Y) <- groups2(bf, cs)(xs, ys), y <- Y}

= join { (x, y) | x <- xs, y <- ys, cs(y, x) }

Synchrony iterator

syncGenGrp is somewhat ugly when extended to multiple collections

Decompose it into Synchrony iterator

syncGenGrp(bf, cs)(xs, ys) =

{

val yi = new Eiterator(ys, bf, cs);

for (x <- xs)

yield (x, yi.syncedWith(x))

}

```
// 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())
  }
}
```

Simultaneous synchronized iteration on multiple collections

Eiterator is convenient to add to function libraries in any popular programming languages, w/o changing any of their compilers

But if you can touch the compilers, things get even more appealing…

Introduce a new 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)
```

Compile it as

```
yi_1 = new EIterator(ys_1, bf_1, cs_1); ...;
yi_n = new EIterator(ys_n, bf_n, cs_n);
x <- xs;
zs_1 = yi_1.syncedWith(x); ...;
zs_n = yi_n.syncedWith(x);
```

Example

```
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)
```

$O(|ws|(|xs| + k|ys| + k^2|zs| + k^3))$, assuming no event overlaps more than k other events

```
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
  val zi = new EIterator(zs, isBefore
  for (
    w <- ws;
    x <- xi.syncedWith(w);
    y <- yi.syncedWith(w);
    z <- zi.syncedWith(w);
    s = max(w.start, x.start, y.start
    e = min(w.end, x.end, y.end, z.enc
    if s < e
  ) yield Event(start = s, end = e, ic
}
```

$O((k3 + 1)|ws| + 2k(|xs| + |ys| + |zs|))$, which is linear when k is small

```
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)
}
```

GMQL emulation, a stress test

GMQL is an advanced genomic query system

Handles complex non-equijoins on genomic regions

GMQL ~24k lines of codes

Synchrony emulation ~4k lines, much faster, needs much less memory

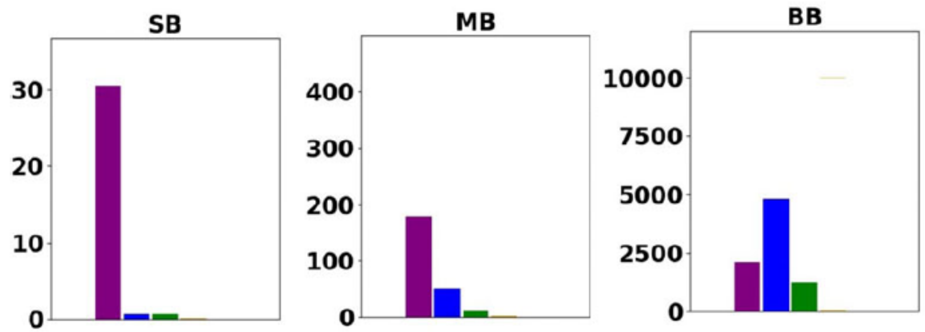


Fig. 13. Performance of GMQL CLI and Synchrony emulation on simple region MAP. Time in seconds, average of 30 runs for SB and MB, and 5 runs for BB. *Purple:* GMQL CLI. *Blue:* Sequential Synchrony emulation. *Green:* Sample-parallel Synchrony emulation.

The GMQL MAP query is emulated using a Synchrony iterator like this:

```
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)
}
```

Summary

Synchrony generator & iterator

A programming pattern for synchronized iteration

A conservative extension of comprehension syntax in a 1st-order restricted setting

Generalization of efficient relational database merge join to antimonotonic predicates

See our paper (*JFP*, 32:e9, 2022) for details ☺