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 $O(|xs| \cdot |ys|)$

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

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

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: zs 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, zs, 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) \mid x, y \in \text{taxpayers}, x \text{ earns less but pays more tax than } y \} \\
\{ (x,y) \mid x, y \in \text{mobile phones}, x\text{’s price is similar to } y\text{’s price} \}
Limited mixing lemma

Let $e(X)$ be an expression in $\text{NRC}_1(<)$ and $e[C/X][C’]$. Suppose $e(X)$ has at most linear-time complexity wrt size of $X$. Then for each $(u,v)$ in $\text{gaifman}(C’)$, either

$(u,v)$ in $\text{gaifman}(C)$, or

$u$ in $\text{atom}^0(C)$ and $v$ in $\text{atom}^1(C)$, or

$u$ in $\text{atom}^1(C)$ and $v$ in $\text{atom}^0(C)$

Similar limited mixing lemmas can be proved for

$\text{NRC}_1(\text{takewhile}, \text{dropwhile}, \text{sort}, <)$

$\text{NRC}_1(\text{foldleft}, \text{sort}, <)$

$\text{NRC}_1(\text{zip}, \text{sort}, <)$
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 \((x \preceq x' | xs)\), then \(\forall y \in ys\): \(bf(y, x)\) implies \(bf(y, x')\)

If \((y' \preceq y | ys)\), then \(\forall x \in xs\): \(bf(y, x)\) implies \(bf(y', x)\)

**Antimonotonicity of \(cs\) wrt \(bf\)**

If \((x \preceq x' | xs)\), then \(\forall y \in ys\): \(bf(y, x) \& \neg cs(y, x)\) implies \(\neg cs(y, x')\)

If \((y' \preceq y | xs)\), then \(\forall x \in xs\): \(\neg bf(y, x) \& \neg cs(y, x)\) implies \(\neg cs(y', x)\)
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`:

\[
\text{ov1}(xs, ys) = \text{ov4}(xs, ys)
\]

\[
\text{ov1}(xs, ys) \text{ has complexity } O(|xs| \cdot |ys|)
\]

\[
\text{ov2}(xs, ys) \text{ has complexity } O(|xs| + k|ys|),
\]

where each event in `ys` overlaps fewer than `k` events in `xs`
**syncGenGrp is a conservative extension**

The functions definable in $\text{NRC}_1(<)$ and $\text{NRC}_1(<, \text{syncGenGrp})$ 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

<table>
<thead>
<tr>
<th>type</th>
<th>form</th>
<th>usual implementation</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>equijoin</td>
<td>x.a = y.b</td>
<td>hash join, merge join</td>
<td>convex, reflexive</td>
</tr>
<tr>
<td>single inequality</td>
<td>x.a ≤ y.b</td>
<td>merge join</td>
<td>Convex, reflexive</td>
</tr>
<tr>
<td>range join</td>
<td>x.a – e ≤ y.b ≤ x.a + e</td>
<td>range join</td>
<td>Convex, reflexive</td>
</tr>
<tr>
<td>band join</td>
<td>x.a ≤ y.b ≤ x.c</td>
<td>band join</td>
<td>Convex, reflexive</td>
</tr>
<tr>
<td>interval join</td>
<td>x.a ≤ y.b &amp;&amp; y.c ≤ x.d</td>
<td>Union of two band joins, interval joins for special data types</td>
<td>Non-convex, antimonotonic</td>
</tr>
</tbody>
</table>

Convexity ⇒ antimonotonicity

∴ syncGenGrp implements them simply and efficiently, viz. Synchrony join
**syncGenGrp** generalizes relational merge join from equijoin to antimonotonic predicates.

```scala
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 maybe = ys.takeWhile(y => bf(y, x) || cs(y, x))  
      val yes = maybe.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  
  }

groups = merge join algo, implements relational join when cs is an equijoin predicate  
{(x, y) | x < xs, (x, y) < groups(bf, cs)(xs, ys), y <= Y}  
= join { (x, y) | x < xs, y <= ys, cs(y, x) }  

groups2 = syncGenGrp extensionally & intensionally  

groups2 = a novel “synchrony” join algo, implements relational join when cs is an antimonotonic predicate  
{(x, y) | x < xs, (x, 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

```scala
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

\[(x, z_{s1}, ..., z_{sn}) \leftarrow xs \text{ syncWith}(ys_1, bf_1, cs_1) \ldots \text{ syncWith}(ys_n, bf_n, cs_n)\]

Compile it as

\[
y_{i1} = \text{new } \text{EIterator}(ys_1, bf_1, cs_1); \ldots;
y_{in} = \text{new } \text{EIterator}(ys_n, bf_n, cs_n);
x \leftarrow xs;
z_{s1} = y_{i1}.\text{syncedWith}(x); \ldots;
z_{sn} = y_{in}.\text{syncedWith}(x);
\]
Example

```python
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, 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)
    }
}
```

```python
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 syncedWith(xs, isBefore, overlap)
        syncedWith(ys, isBefore, overlap)
        syncedWith(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)
    }
}
```

\[
O(|ws|(|xs| + k|ys| + k^2|zs| + k^3)),
\]

assuming no event overlaps more than k other events.

\[
O((k^3 + 1)|ws| + 2k(|xs| + |ys| + |zs|)),
\]

which is linear when k is small.
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

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

```scala
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 1\textsuperscript{st}-order restricted setting*

*Generalization of efficient relational database merge join to antimonotonic predicates*

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