From comprehension syntax to efficient non-equijoins: A journey with Val Tannen

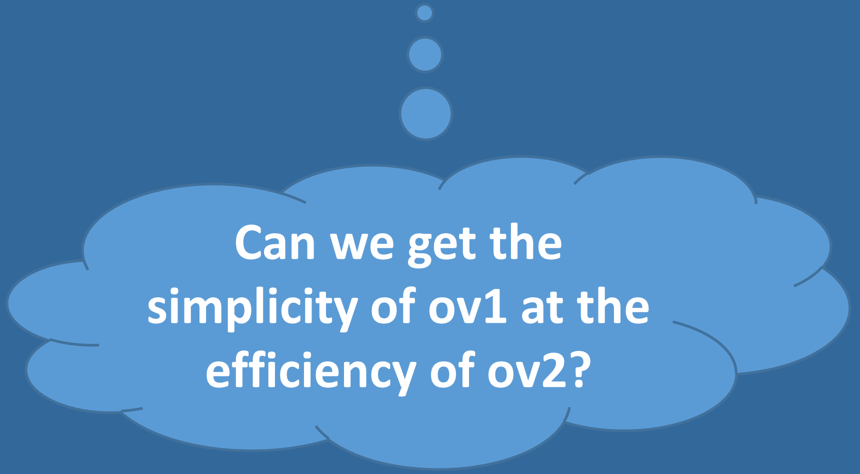
Limsoon Wong

Motivating example

If xs and ys are sorted according to isBefore, then 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




```
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())
}
```

Is there an 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 ∈ taxpayers, x earns less but pays more tax than y }

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

Comprehension syntax in a 1st order setting

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 $\mathcal{NRC}_1$

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

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

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

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

Call-by-value Operational semantics

Time complexity of a node

time(e⇓C) = 1 + # branches of the node

Time complexity of an evaluation tree

time(e⇓) = sum of time complexity of all nodes in the evaluation tree

Note: time(C⇓C) = 1

$$\frac{}{C \Downarrow C}$$

$$\frac{e_1 \Downarrow C_1 \quad \ldots \quad e_n \Downarrow C_n}{(e_1, \ldots, e_n) \Downarrow (C_1, \ldots, C_n)} \qquad \frac{e \Downarrow (C_1, \ldots, C_n)}{e.\pi_i \Downarrow C_i}\, 1 \leq i \leq n$$

$$\frac{}{\{\} \Downarrow \{\}} \qquad \frac{e \Downarrow C}{\{e\} \Downarrow \{C\}} \qquad \frac{e_1 \Downarrow C_1 \quad e_2 \Downarrow C_2}{e_1 \cup e_2 \Downarrow C_1 \oplus C_2}$$

$$\frac{e_2 \Downarrow \{C_1, \ldots, C_n\} \quad e_1[C_1/x] \Downarrow C_1' \quad \cdots \quad e_1[C_n/x] \Downarrow C_n'}{\bigcup\{e_1 \mid x \in e_2\} \Downarrow C_1' \oplus \cdots \oplus C_n'}$$

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

$$\frac{e_1 \Downarrow \mathit{true} \quad e_2 \Downarrow C}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow C} \qquad \frac{e_1 \Downarrow \mathit{false} \quad e_3 \Downarrow C}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow C}$$

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

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

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

Polynomiality of $NRC_1(<)$

Let $e(X_1, \ldots, X_n)$ be an expression in $NRC_1(<)$. Then there is a number k such that the time complexity of $e(X_1, \ldots, X_n)$ is $\Theta(n^k)$

I.e., if the time complexity of $e(X_1, \ldots, X_n)$ is sub-quadratic, it must be either linear or constant time; and if it is sub-linear, it must be constant time

Furthermore, these properties are retained when NRC1(<) is augmented by any additional functions that have polynomial time complexity

Limited mixing lemma

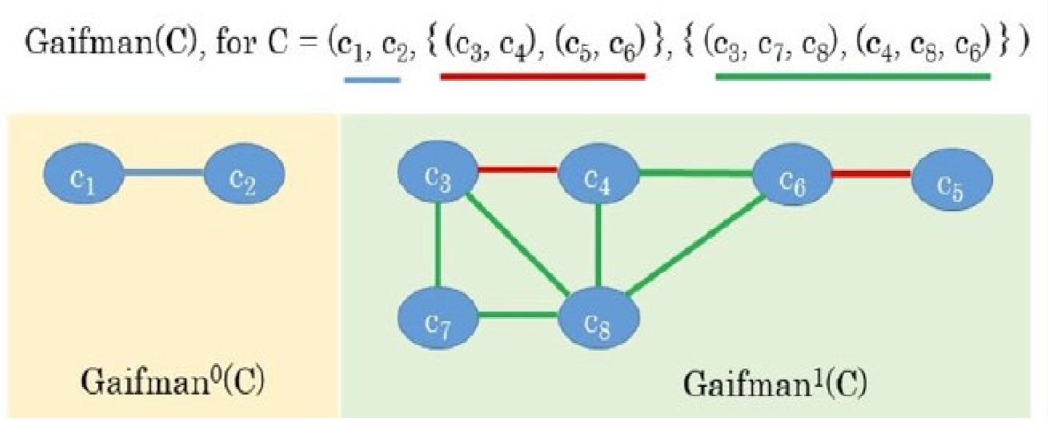
Let e(X) be an expression in $NRC_1(<)$ and e[C/X]⇓C'. Suppose e(X) has at most linear-time complexity wrt size of X. Then for each (u,v) in Gaifman(C'), either

(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 $atom^0(C)$

Proof: See my festschrift paper



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

There is an intensional expressiveness gap

Zip(X,Y): $\{ b_1 \times b_2 \}$ is an expression in NRC1(<) where X: $\{ b_3 \times b_1 \}$ and Y: $\{ b_3 \times b_2 \}$, such that Zip(X,Y) $\Downarrow \{ (u_1, v_1), \ldots, (u_n, v_n) \}$ for every X == $\{ (o_1, u_1), \ldots, (o_n, u_n) \}$ and Y == $\{ (o_1, v_1), \ldots, (o_n, v_n) \}$, $o_1, \ldots, o_n$ distinct

Zip is a low-complexity join. But time complexity in $NRC_1(<)$ is $\Omega(|U| \cdot |V|)$

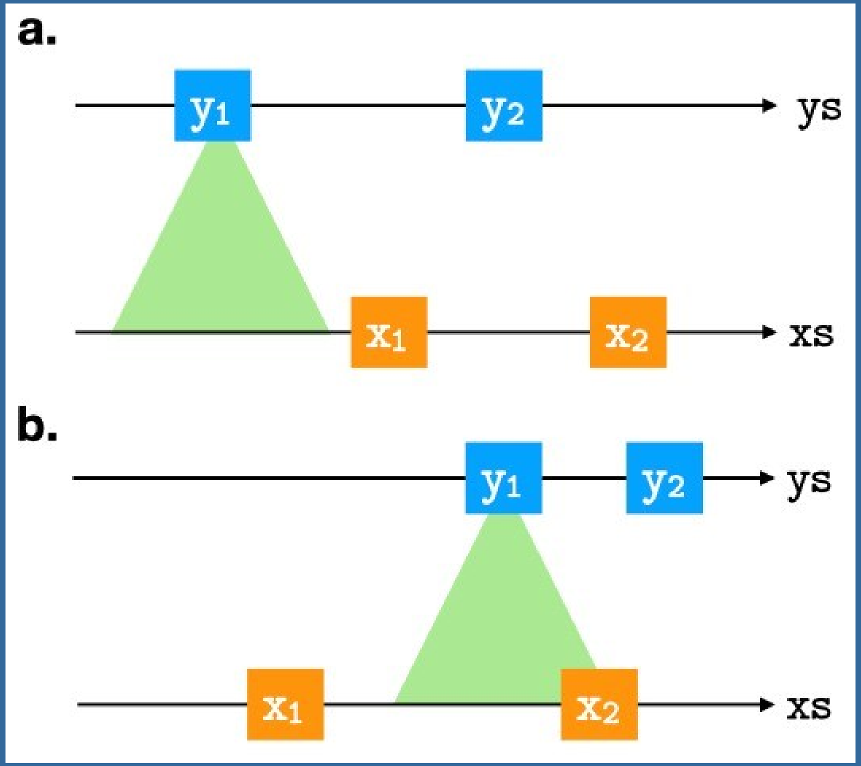
Proof sketch: Gaifman($\{ (u_1, v_1), \ldots, (u_n, v_n) \}$) = $\{ (u_1, v_1), \ldots, (u_n, v_n) \}$. Suppose Zip has at most linear time complexity. As $(u_i, v_i) \notin$ gaifman(U,V) = $U \cup V$, by Limited Mixing Lemma, either u_i or v_i is in $atom^0(U,V)$. But $atom^0(U) = atom^0(V) = \{ \}$. A contradiction.

How to fill the gap?

What new library function or programming construct fills this intensional expressiveness 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)

Right-side convexity

Synchrony generator, capturing a 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 of $NRC_1(<)$

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

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

Thus, syncGenGrp fills the intensional expressive power gap of comprehension syntax in a “1^{st}-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 }

= { (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 = “synchrony” join algo, implements relational join when cs is an antimonotonic predicate

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

= { (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())
  }
}
```

Synchrony iterator, with simple cache

```
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 }
  }
}
```

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, zs₁, ..., zsₙ) <- xs syncWith(ys₁, bf₁, cs₁) ...
                         syncWith(ysₙ, bfₙ, csₙ)
```

Compile it as

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

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

```
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)
}
```

$O(k^3|ws| + k(|xs|+|ys|+|zs|))$
which is linear when k is small

GMQL emulation, a stress test

GMQL is a genomic query system developed by Stefano Ceri

Handles complex non-equijoins on genomic regions

~24k lines of codes

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

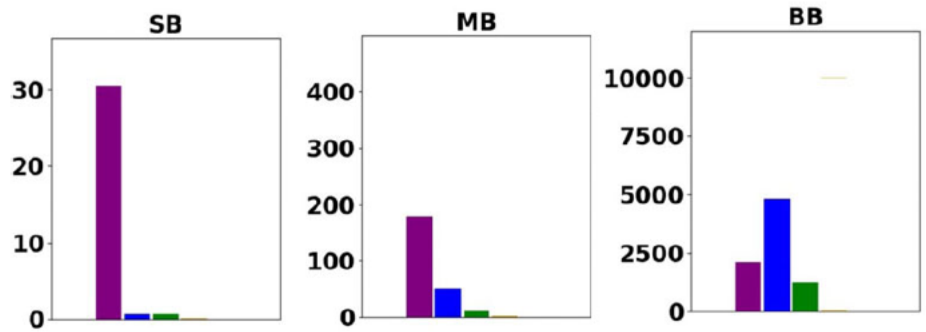


Fig. 13. Performance of GMQL CLI and Synchrony emulation on simple region `MAP`. Time in seconds, average of 30 runs for SB and MB, and 5 runs for BB. *Purple:* GMQL CLI. *Blue:* Sequential Synchrony emulation. *Green:* Sample-parallel Synchrony emulation.

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

There is indeed an intensional expressiveness gap of using comprehension syntax as querying bulk data types

Synchrony iterator rescues comprehension syntax from this gap

A programming pattern for synchronized iteration

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

Generalization of efficient relational database merge join to antimonotonic (non-equijoin) predicates

References

Limsoon Wong, “An intensional expressiveness gap of comprehension syntax”, *OASIcs* 119:???. Tannen’s Festschrift. In press.

Stefano Perna, Val Tannen, Limsoon Wong, “Iterating on multiple collections in synchrony”, *JFP* 32:e9, 2022. doi:10.1017/S0956796822000041