From comprehension syntax to efficient non-equi-joins: 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 $O(|xs| \cdot |ys|)$

$ov2(xs, ys)$ 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 $ov2$?

case class Event(start: Int, end: Int, id: String)
  // Constraint: start < end
val isBefore = (y: Event, x: Event) => {
  if (y.start < x.start) {
  } else if (y.start == x.start && y.end < x.end) {
  } else if (y.start <= x.start && x.end <= y.end) {
  } else if (y.start < x.start && y.end > x.end) {
  } else if (y.start <= x.start && x.end < y.end) {
  } else if (y.start <= x.start && x.end <= y.end) {
  } else if (y.start > x.start && y.end <= x.end) {
  } else if (y.start > x.start && y.end > x.end) {
  } else {
  }
}

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

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(x: Event, y: Event, zs: Vec[Event], acc: Vec[(Event, Event)]) = {
    // Key Invariant: aux(xs, ys, Vec(), acc) = acc ++ ov1(xs, ys)
    aux(x, y, zs, if (xs.isEmpty) acc
      else if (zs.isEmpty && ys.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 (false, true) => aux(xs.tail, zs ++ (ys, Vec()), acc)
          case (_, true) => aux(xs.tail, zs ++ (ys, Vec()), acc)
        }
      }
    }
  }
  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) \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's \text{ price is similar to } y's \text{ price} \}\}
Comprehension syntax in a 1st order setting

Types in $\mathcal{NRC}_1$

\[
\begin{align*}
  t &::= b \mid b_1 \times \cdots \times b_n \\
  s &::= t \mid \{t\} \mid s_1 \times \cdots \times s_n
\end{align*}
\]
where $b$’s are base types.

Expressions in $\mathcal{NRC}_1$

\[
\begin{align*}
  C^i : s & \quad x^i : s & \quad e_1 : b_1 \ldots e_n : b_n & \quad e : b_1 \times \cdots \times b_n \quad 1 \leq i \leq n \\
  (e_1, \ldots, e_n) : b_1 \times \cdots \times b_n & \quad e.\pi_i : b_i
\end{align*}
\]

\[
\begin{align*}
  \{\} : \{t\} & \quad \{e\} : \{t\} & \quad e_1 \cup e_2 : \{t\} & \quad e_1 : \{t_1\} \quad e_2 : \{t_2\} \\
  \bigcup \{e_1 \mid x^i \in e_2\} : \{t_1\}
\end{align*}
\]

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

$e_1 : s$  $e_2 : s$  $e_1 < e_2 : \mathbb{B}$  $e_1 = e_2 : \mathbb{B}$  $e : \{t\}$  $e \text{ isempty} : \mathbb{B}$
Call-by-value
Operational semantics

Time complexity of a node
\[
\text{time}(e_{\downarrow} C) = 1 + \# \text{ branches of the node}
\]

Time complexity of an evaluation tree
\[
\text{time}(e_{\downarrow}) = \text{sum of time complexity of all nodes in the evaluation tree}
\]

Note: \(\text{time}(C_{\downarrow} C) = 1\)
Polynomiality of $\text{NRC}_1(\langle\rangle)$

Let $e(X_1, \ldots, X_n)$ be an expression in $\text{NRC}_1(\langle\rangle)$. 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 $\text{NRC}1(\langle\rangle)$ is augmented by any additional functions that have polynomial time complexity
Limited mixing lemma

Let $e(X)$ be an expression in $\text{NRC}_1(\langle \rangle )$ and $e[C/X] \upharpoonright C'$. 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 $\text{atom}^0(C)$

Proof: See my festschrift paper
There is an intensional expressiveness gap

Zip(X,Y): \{ b_1 \times b_2 \} is an expression in NRC1(\textless) 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(\textless) 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 \( (u_i, v_i) \notin \text{gaifman}(U,V) = U \cup V \), by Limited Mixing Lemma, either \( u_i \) or \( v_i \) is in \( \text{atom}^0(U,V) \). But \( \text{atom}^0(U) = \text{atom}^0(V) = \{ \} \). 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 $(x \preceq x' \mid xs)$, then $\forall y \in ys: bf(y, x)$ implies $bf(y, x')$

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

Antimonotonicity of $cs$ wrt $bf$

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

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

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 $O(|xs| \cdot |ys|)$, where each event in $ys$ overlaps fewer than $k$ events in $xs$.

$$ov2(xs, ys)$$

has complexity $O(|xs| + k \cdot |ys|)$.

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 $O(|xs| \cdot |ys|)$, where each event in $ys$ overlaps fewer than $k$ events in $xs$.

$$ov2(xs, ys)$$

has complexity $O(|xs| + k \cdot |ys|)$.

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 $O(|xs| \cdot |ys|)$, where each event in $ys$ overlaps fewer than $k$ events in $xs$.

$$ov2(xs, ys)$$

has complexity $O(|xs| + k \cdot |ys|)$.

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 $O(|xs| \cdot |ys|)$, where each event in $ys$ overlaps fewer than $k$ events in $xs$.

$$ov2(xs, ys)$$

has complexity $O(|xs| + k \cdot |ys|)$.
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 (non-equi) 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>$x.a \leq y.b$</td>
<td>merge join</td>
<td>Convex, reflexive</td>
</tr>
<tr>
<td>range join</td>
<td>$x.a - e \leq y.b \leq x.a + e$</td>
<td>range join</td>
<td>Convex, reflexive</td>
</tr>
<tr>
<td>band join</td>
<td>$x.a \leq y.b \leq x.c$</td>
<td>band join</td>
<td>Convex, reflexive</td>
</tr>
<tr>
<td>interval join</td>
<td>$x.a \leq y.b$ &amp;&amp; $y.c \leq x.d$ where $x.a \leq x.d$ and $y.c \leq y.b$</td>
<td>Union of two band joins, interval joins for special data types</td>
<td>Non-convex, antimonotonic</td>
</tr>
</tbody>
</table>

Convexity $\Rightarrow$ antimonotonicity

$\therefore$ syncGenGrp implements them simply and efficiently, viz. Synchrony join
syncGenGrp generalizes relational merge join from equijoin to antimonotonic predicates

\[
\text{groups} = \text{merge join algo, implements relational join when}\ cs \text{ is an equijoin predicate}
\]

\[
\{ (x, y) \mid x \leftarrow xs, (\_,Y) \leftarrow \text{groups}(bf, cs)(xs, ys), y \leftarrow Y \}
\]

\[
= \{ (x, y) \mid x \leftarrow xs, y \leftarrow ys, cs(y, x) \}
\]

\[
\text{def groups[A,B]}
\]

\[
(\text{bf: (B,A) \Rightarrow Boolean, cs: (B,A) \Rightarrow Boolean})
\]

\[
(xs: \text{Vec[A]}, ys: \text{Vec[B]})
\]

\[
: \text{Vec[(A,Vec[B])]} = \{
\]

\[
\text{def step(acc: (Vec[(A,Vec[B])], Vec[B]), x: A)}
\]

\[
: (\text{Vec[(A, Vec[B])], Vec[B]} = \{
\]

\[
\text{val} (xzss, ys) = \text{acc}
\]

\[
// \text{this works only for equijoin cs:}
\]

\[
\text{val} yt = ys.dropWhile(y \Rightarrow bf(y, x))
\]

\[
// \text{this works for convex cs:}
\]

\[
\text{val} yt = ys.dropWhile(y \Rightarrow bf(y, x) \land cs(y, x))
\]

\[
\text{val} zs = yt.takeWhile(y \Rightarrow cs(y, x))
\]

\[
(xzss :+ (x, zs), yt)
\]

\[
\}
\]

\[
\text{val e: (Vec[(A,Vec[B])], Vec[B]) = (Vec(), ys)}
\]

\[
\text{val} (xzss, _) = xs.foldLeft(e)(\text{step } _)\]

\[
\text{return} xzss\]

\[
\]

\[
\text{def groups2[A,B]}
\]

\[
(\text{bf: (B,A) \Rightarrow Boolean, cs: (B,A) \Rightarrow Boolean})
\]

\[
(xs: \text{Vec[A]}, ys: \text{Vec[B]})
\]

\[
: \text{Vec[(A,Vec[B])]} = \{
\]

\[
// \text{Requires: bf monotonic wrt (zs, ya); cs antimonotonic wrt bf.}
\]

\[
\text{val} step = (\text{acc: (Vec[(A,Vec[B])], Vec[B]), x: A) \Rightarrow \{
\]

\[
\text{val} (xzss, ys) = \text{acc}
\]

\[
\text{val} maybees = ys.takeWhile(y \Rightarrow bf(y, x) \lor cs(y, x))
\]

\[
\text{val} yeses = ys.filter(y \Rightarrow cs(y, x))
\]

\[
\text{val} nos = ys.dropWhile(y \Rightarrow bf(y, x) \land cs(y, x))
\]

\[
(xzss :+ (x, yeses), yes ++: nos)
\]

\[
\}
\]

\[
\text{val} e: (\text{Vec[(A,Vec[B])], Vec[B]) = (Vec(), ys)}
\]

\[
\text{val} (xzss, _) = xs.foldLeft(e)(\text{step})\]

\[
\text{return} xzss\]

\[
\]

\[
\text{groups2} = \text{syncGenGrp extensionally & intensionally}
\]

\[
\]

\[
\text{groups2} = \text{“synchrony” join algo, implements relational join when}\ cs \text{ is an antimonotonic predicate}
\]

\[
\{ (x, y) \mid x \leftarrow xs, (\_,Y) \leftarrow \text{groups2}(bf, cs)(xs, ys), y \leftarrow Y \}
\]

\[
= \{ (x, y) \mid x \leftarrow xs, y \leftarrow ys, cs(y, x) \}
\]
Synchrony iterator

class EIterator[A,B](
  eles: Vec[B],
  bf: (B,A)=>Boolean, cs:(B,A)=>Boolean) {
  private var es = eles

  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())
  }
}

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))
}
class Iterator[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 σ

  // 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 σ is same as the last σ
    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

\[
(x, zs_1, \ldots, zs_n) \leftarrow xs \text{ syncWith}(ys_1, bf_1, cs_1) \ldots \text{ syncWith}(ys_n, bf_n, cs_n)
\]

Compile it as

\[
\begin{align*}
y_{i_1} &= \text{new EIterator}(ys_1, bf_1, cs_1); \ldots; \\
y_{i_n} &= \text{new EIterator}(ys_n, bf_n, cs_n); \\
x &= \text{xs;} \\
zs_1 &= y_{i_1}.\text{syncedWith}(x); \ldots; \\
zs_n &= y_{i_n}.\text{syncedWith}(x);
\end{align*}
\]
Example

\[ O(|ws| \cdot |xs| \cdot |ys| \cdot |zs|) \]

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

which is linear when k is small
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

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)
  }
```
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-eqijoin) predicates