Fair intersection of seekable iterators

MICHAEL ARNTZENIUS, UC Berkeley, USA

miniKanren's key semantic advance over Prolog is to implement a complete yet efficient search strategy, fairly interleaving execution between disjuncts. This fairness is accomplished by bounding how much work is done exploring one disjunct before switching to the next. We show that the same idea—fairness via bounded work—underlies an elegant compositional approach to implementing worst-case optimal joins using a seekable iterator interface, suitable for shallow embedding in functional languages.

ACM Reference Format:
Michael Arntzenius. 2025. Fair intersection of seekable iterators. 1, 1 (October 2025), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Our approach and its scope

This paper originates in early work towards the implemention of a relational programming language inspired by Datalog and modern database query engine techniques, in particular, worst-case optimal joins [Ngo et al. 2012] and specifically Leapfrog Triejoin [Veldhuizen 2014]. Join algorithms often have set intersection algorithms at their core. In implementing the 'leapfrog' intersection of Leapfrog Triejoin, we discovered it was not compositional—a k-way intersection cannot be decomposed into nested binary intersections without loss of performance.

We show how to restore compositionality by borrowing a trick from miniKanren's complete disjunctive search. This permits shallowly embedding efficient multi-way set intersection in a functional language. Although we do not describe it here, we have extended this to full conjunctive queries. In future we hope to extend it to full-fledged logic programming. We present our work in Haskell because algebraic data types clarify its presentation and laziness is convenient for representing potentially infinite search spaces. Neither is essential: any disciplined programmer can observe a typing discipline; any language with closures, objects, or similar features can encode laziness.

2 Fair union of streams

When multiple rules apply to the current goal, Prolog tries them in order, exploring each to exhaustion before starting the next. If exploring a rule fails to terminate, later rules are not visited, and potential solutions they might generate are lost. If we think of goal-directed search as yielding a stream of solutions, Prolog implements disjunction between rules as stream concatenation. In Haskell, using laziness to represent possibly-infinite streams, we might implement this like so:

```
data Stream a = Empty
              | Yield a (Stream a)
append Empty        ys = ys
append (Yield x xs) ys = Yield x (append xs ys)          -- keep focus on xs
```

Author's Contact Information: Michael Arntzenius, daekharel@gmail.com, UC Berkeley, Berkeley, CA, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2025/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

As we've hinted, concatenation is an incomplete search strategy: `append xs ys` will never visit `ys` if `xs` is infinite. We can fix this by interleaving `xs` and `ys` instead:

```
interleave Empty        ys = ys
interleave (Yield x xs) ys = Yield x (interleave ys xs) -- swap focus to ys
```

However, this is complete only if both streams are *productive*—they must not loop indefinitely without producing either `Empty` or `Yield`. Unfortunately, simple and desirable operations on lazy lists may not preserve productivity. For instance, a productive infinite list becomes unproductive if filtered to a finite subset. Consider the even prime numbers:

```
filter :: (a -> Bool) -> Stream a -> Stream a {- definition omitted -}
primes = Yield 2 (Yield 3 (Yield 5 ...)) {- full definition omitted -}
evenPrimes = filter even primes
twoThreeFour = interleave evenPrimes (Yield 3 (Yield 4 Empty))
```

Here, `evenPrimes` is equivalent to `Yield 2 ⊥`, where ⊥ is an unproductive infinite loop. Because of this, `twoThreeFour` is equivalent to `Yield 2 (Yield 3 ⊥)`; the call to `interleave` gets stuck evaluating the ⊥ in `evenPrimes` before it can yield 4.[1]

To find a complete stream union operator, we make an analogy to *fairness* in concurrent programming, which means that "every process gets a chance to make progress, regardless of what other processes do" [Owicki and Lamport 1982], although it has no singular formal definition [Kwiatkowska 1989; Völzer et al. 2005]. We can imagine streams as processes that make progress toward finding solutions when evaluated. In this analogy, stream union is a 2-to-1 multiplexing scheduler that allocates the time given to it between two sub-processes. Completeness requires fairness: one stream may not permanently block the other from making progress. Interleaving *elements* between streams is not sufficient, because finding the next element (or determining none exists) can require an infinite amount of work, blocking execution of the other stream.

We therefore need all streams to yield control after doing at most finite work. Under lazy evaluation, we yield control when we produce a constructor. Therefore, we extend `Stream` with a constructor `Later` that yields control to the consumer without giving further information:

```
data Stream a = Empty
              | Yield a (Stream a)
              | Later   (Stream a)
```

`Later` corresponds to the "immature" streams of μKanren [Hemann and Friedman 2013], which it implements as thunks; as Haskell is lazy, we need not explicitly thunk them. As in μKanren, we actually ensure a stronger property, which we will call `Later`-productivity: all streams do finite work before either terminating or yielding `Later`, disallowing unbroken infinite repetition of `Yield`. We can ensure this property conservatively by guarding all recursion with `Later`.[2] For instance, we must redefine `primes` to be equivalent to `Yield 2 (Later (Yield 3 (Later (Yield 5 ...))))` or similar—the exact frequency of `Later` is unimportant so long as it occurs infinitely often.

Using this, we implement fair stream union (μKanren's `mplus`):

[1] This incompleteness afflicts the work of Kiselyov et al. [2005]. Their `interleave` is in effect the CPS translation of ours. We implemented `twoThreeFour` at type `SFKT Identity Int` and found that `runIdentity (solve twoThreeFour)` returned `2:3:⊥`. More directly, if we let `evenOdds = odds >>- \x -> if even x then return x else mzero`, then `observe (evenOdds `interleave` return 0)` is ⊥, but `observe (return 0 `interleave` evenOdds)` is 0.

[2] In many cases it is possible to be less conservative, for instance in our definitions of `union` and `filter` below. There is a long line of work on type systems that ensure productivity by checking recursion is appropriately guarded which could be applied here, including Nakano [2000], Birkedal et al. [2012], and Atkey and McBride [2013].

```
union Empty        ys = ys
union (Yield x xs) ys = Yield x (union xs ys)  -- keep focus on xs
union (Later xs)   ys = Later   (union ys xs)  -- swap focus to ys
```

as well as filter (related to but less expressive than μKanren's `bind`):

```
filter p Empty = Empty
filter p (Yield x xs) = if p x then Yield x xs' else xs'
  where xs' = filter p xs
filter p (Later xs) = Later (filter p xs)
```

Note that `union` and `filter` preserve **`Later`**-productivity of streams. Note also that in our concurrency analogy, `union` acts as a cooperative scheduler, using **`Later`** as a signal to switch processes. It's been observed in miniKanren that the placement of immature streams in a program greatly affects performance; this is because it determines the scheduling of (effectively) a massively concurrent program.

Using `filter` and `union`, we can define `twoThreeFour` completely:

```
twoThreeFour = union (filter even primes) (Yield 3 (Yield 4 Empty))
```

If we assume `primes` is `Yield 2 (Later ...)`, then `twoThreeFour` is equivalent to `Yield 2 (Later (Yield 3 (Yield 4 ⊥)))`, where color indicates source: `filter even primes` generates the the blue terms `Yield 2`, `Later`, `⊥`, while `Yield 3 (Yield 4 Empty)` generates the red terms `Yield 3`, `Yield 4`.

Complete disjunction, then, is accomplished by bounding the work a stream does before handing control to the next stream. In the rest of this paper, we apply a similar trick to intersection rather than union. In section 3 we show that the leapfrog intersection of seekable iterators used in the worst-case optimal join algorithm Leapfrog Triejoin [Veldhuizen 2014] is *not* fair. To remedy this, in section 4 we show how to extend the seekable iterator interface to allow *bounded interleaving* between sub-iterators.

3 Unfair intersection of seekable iterators

Suppose we have a collection of key-value pairs, stored in sorted order so that we can efficiently seek forward to find the next pair whose key satisfies some lower bound by e.g. galloping search [Bentley and Yao 1976]. We can capture these assumptions in a *seekable iterator* interface:

```
data Iter k v = Empty
              | Yield k v (k -> Iter k v)
```

A seekable iterator, **`Iter`** `k v`, is like a stream of key-value pairs, **`Stream`** `(k,v)`, except that (a) it yields pairs in ascending key order, and (b) rather than the entire remainder of the stream, **`Yield`** produces a function `seek :: k -> Iter k v` such that `seek target` iterates over the remaining pairs with keys `k >= target`. To recover the entire remainder, we pass `seek` the just-visited key; this lets us iterate over all elements of the stream:

```
toSorted :: Iter k v -> [(k, v)]
toSorted Empty = []
toSorted (Yield k v seek) = (k,v) : toSorted (seek k)
```

We can easily turn a sorted list `[(k, v)]` into a seekable iterator, although seeking will not be efficient since Haskell lists allow only linear, in-order access. We could use a more appropriate data structure, such as a sorted array or balanced tree, but omit this as it is not crucial to our explanation:

```
fromSorted :: Ord k => [(k, v)] -> Iter k v
```

```
fromSorted [] = Empty
fromSorted ((k,v) : rest) = Yield k v seek
  where seek k' = fromSorted (dropWhile ((< k') . fst) rest)
```

We can intersect two seekable iterators by leapfrogging: repeatedly advance the iterator at the smaller key towards the higher, until either both iterators reach the same key or one is exhausted:

```
intersect :: Ord k => Iter k a -> Iter k b -> Iter k (a,b)
intersect Empty _ = Empty
intersect _ Empty = Empty
intersect s@(Yield k1 x s') t@(Yield k2 y t') =
  case compare k1 k2 of
    LT -> intersect (s' k2) t -- s < t, so seek s toward t
    GT -> intersect s (t' k1) -- t < s, so seek t toward s
    EQ -> Yield k1 (x,y) (\k' -> intersect (s' k') (t' k'))
```

However, `intersect`'s performance can suffer asymptotically when intersecting more than two iterators. For instance, consider three subsets of the integers between 0 and 7,777,777—the evens, the odds, and the endpoints:

```
evens = fromSorted [(x, "even") | x <- [0, 2 .. 7_777_777]]
odds  = fromSorted [(x, "odd")  | x <- [1, 3 .. 7_777_777]]
ends  = fromSorted [(x, "end")  | x <- [0,      7_777_777]]
```

The intersection of `evens` and `odds`, and therefore of all three sets, is empty. We can compute this by calling `intersect` twice, but performance improves dramatically if, rather than intersecting `evens` and `odds` first, we intersect one of them with `ends` first. At the GHCi repl:

```
ghci> -- set +s to print time statistics
ghci> :set +s
ghci> toSorted ((evens `intersect` odds) `intersect` ends)
[]
(5.57 secs, 5,288,958,128 bytes)
ghci> toSorted (evens `intersect` (odds `intersect` ends))
[]
(0.57 secs, 248,961,040 bytes)
```

The reason is simple: "leapfrogging" `evens` and `odds` against one another performs a full linear scan of both lists. Whatever our current key x in `even` (e.g. $x = 1$), we seek forward past x in `odds` and reach $x + 1$; then we seek past $x + 1$ in `even` to $x + 2$, and so on. We do 7,777,777 units of work before we determine the intersection is empty. By contrast, intersecting `odds` with `ends` almost immediately skips to the end of both relations. (This occurs even though we are *not* materializing any intermediate results.)[3]

The problem is that `intersect` does not—and with our **`Iter`** interface, *cannot*—fairly interleave work between its arguments. Instead, like `interleave`, it blocks first on one, then the other. Our leapfrogging `intersect` implements the $k = 2$ case of the k-way set intersection of Demaine et al. [2000] and Veldhuizen [2014]. Repeated application of 2-way `intersect` fails to capture the essence of this k-way algorithm, which is *to propagate lower bounds on keys between all intersected*

[3]We are telling a white lie here: because we use Haskell lists, seeking is linear-time, and there is no asymptotic slow-down. We are instead observing the difference between an *interpreted* inner loop (`intersect`, loaded at the GHCi repl) and a *compiled* one (`dropWhile` from the standard library). However, had we used sorted arrays with binary or galloping search, or balanced trees, there would be a true asymptotic speed-up for the reasons we describe.

iterators. Evaluating `(evens `intersect` odds) `intersect` ends` immediately waits for `evens `intersect` odds` to find a key, which takes $O(n)$ work (where n is the size of `evens`/`odds`). This blocks information exchange between `ends` and `evens`/`odds` that would let us jump straight to the end in $O(\log n)$ time.

Can we capture this algorithm compositionally, so that repeated 2-way intersection is as efficient as k-way intersection? Yes: as with stream union, we can side-step the issue by changing our interface to *bound* how much work we do.

4 Fair intersection of seekable iterators

Just as we implemented fair interleaving of streams by allowing a stream *not* to yield an element, we will implement fair interleaving of seekable iterators by allowing them *not* to yield a key-value pair. Applying this lesson naïvely, we might come up with:

```
data Iter' k v = Empty
               | Yield k v (k -> Iter' k v)
               | Later     (k -> Iter' k v)
```

However, this is insufficiently expressive. The essence of leapfrog intersection is to communicate lower bounds between intersected iterators. `Iter'` produces lower bounds only by yielding key-value pairs. Yet if we interrupt a leapfrogging intersection while it is still searching for the next key-value pair, it may still be able to contribute a new lower bound—for instance, while intersecting `evens` and `odds`, after `evens` proposes `k = 0` and `odds` seeks forward to reach `k = 1`, but before we seek again in `evens`, we already know that `k >= 1`. Therefore instead we regard each iterator as having a `Position`, which may be either a key-value pair or a lower bound on future keys:

```
data Position k v = Found k v | Bound (Bound k)
data Bound k = Atleast k | Greater k | Done deriving Eq
```

`Found k v` means we've found a key-value pair `(k,v)`. `Bound (Atleast k)` means all future keys are `>= k`. `Bound Done` means the iterator is exhausted. We will see the purpose of `Greater` shortly.

We can now define the type `Seek` of seekable iterators supporting fair intersection, which possess both a position and a seek function:

```
data Seek k v = Seek
  { posn :: Position k v          -- a key-value pair, or a bound
  , seek :: Bound k -> Seek k v } -- seeks forward toward a bound
```

It simplifies the definition of intersection if seeking is idempotent; thus we require `seek` to consider the remainder of the sequence *including* the current key, rather than dropping it as we did in `Iter`. We must take this into account when defining `toSorted`:

```
toSorted :: Seek k v -> [(k,v)]
toSorted (Seek (Bound Done) _)    = []
toSorted (Seek (Bound p)    seek) = toSorted (seek p)
toSorted (Seek (Found k v)  seek) = (k,v) : toSorted (seek (Greater k))
```

When the iterator has `Found` a pair `(k,v)`, we pass `Greater k` to its seek function to advance *beyond* the key `k`. The idea is that `seek b` seeks towards the first (smallest) key satisfying the bound `b`. We already know that `Atleast lo` is satisfied by keys `k >= lo`. `Greater lo` is satisfied only by *strictly greater* keys, `k > lo`. And `Done` is satisfied by no keys whatsoever. This endows bounds with a natural order: for bounds `p,q` we let `p <= q` iff any key satisfying `q` must satisfy `p`. We can implement this concisely, if nonobviously, as follows:

```
satisfies :: Ord k => Bound k -> k -> Bool
satisfies bound k = bound <= Atleast k
instance Ord k => Ord (Bound k) where
  compare x y = embed x `compare` embed y where
    embed (Atleast k) = (1, Just (k, 1))
    embed (Greater k) = (1, Just (k, 2))
    embed Done        = (2, Nothing)
```

Using `satisfies` we can define `fromSorted`:

```
fromSorted :: Ord k => [(k,v)] -> Seek k v
fromSorted l = Seek posn seek where
  posn = case l of (k,v):_ -> Found k v
                   []      -> Bound Done
  seek target = fromSorted (dropWhile (not . satisfies target . fst) l)
```

To define intersection we need one last helper, which extracts a lower bound on the remaining keys in an iterator:

```
bound :: Seek k v -> Bound k
bound (Seek (Found k _) _) = Atleast k
bound (Seek (Bound p)   _) = p
```

Finally, we can define fair intersection of seekable iterators. If both iterators are at the same key and have found values, we've found an element of the intersection; otherwise we only know there are no keys until after the greater of their bounds. To seek, we simply seek our sub-iterators.

```
intersect :: Ord k => Seek k a -> Seek k b -> Seek k (a,b)
intersect s t = Seek posn' seek' where
  posn' | Found k x <- posn s, Found k' y <- posn t, k == k' = Found k (x, y)
        | otherwise = Bound (bound s `max` bound t)
  seek' k = seek s k `intersect` seek t k
```

The natural next step is to try our evens-odds-ends example. Unfortunately, we now run into the "white lie" noted in footnote 5 in section 3: we cannot expect an asymptotic speedup while we are still using linear-access Haskell lists; the actual effects we saw there were due to interpretation overhead. In a small experiment with Haskell's Arrays package, compiled with `ghc -O`, on arrays of 30 million elements, our naïve **`Iter`** approach from section 3 took 1.2 seconds to enumerate `(evens` ``` `intersect` odds) `intersect` evens ```, while using our **`Seek`** iterators (or reassociating) took less than a millisecond. You can find the benchmarking code for **`Iter`** in figure 1; the code for **`Seek`** is nearly the same, except for an altered `fromSortedArray` method, shown in figure 2.

5 Related and future work

Our intersection algorithm descends from the adaptive set intersection of Demaine et al. [2000] by way of Leapfrog Triejoin [Veldhuizen 2014]. Intersection is a special case of relational join, and the technique described here extends to full conjunctive queries via *nested* intersections. This is the 'trie' in Leapfrog Triejoin: for instance, a finite ternary relation $R \subseteq A \times B \times C$ can be represented as a 3-level trie, i.e. a nested seekable iterator **`Seek`** `a (`**`Seek`** `b (`**`Seek`** `c ()))`. By intersecting on each join column successively, we implement a worst-case optimal join. Worst-case optimal joins therefore seem to be a form of *fair conjunction,* a connection which we are not sure has been noticed previously.

```haskell
import Data.Array
import Text.Printf
import System.CPUTime
import System.IO (hFlush, stdout)

... -- Definitions of `data Iter`, `toSorted`, and `intersect` from earlier in paper.

n = 30_000_000
evens = fromSortedArray [(x, "even") | x <- [0, 2 .. n]]
odds  = fromSortedArray [(x, "odd")  | x <- [1, 3 .. n]]
ends  = fromSortedArray [(x, "end")  | x <- [0,      n]]

fromSortedArray :: Ord k => [(k,v)] -> Iter k v
fromSortedArray l = go 0 where
  arr = listArray (0, hi) l
  hi = length l
  go lo = if lo >= hi then Empty else Yield k v seek where
    (k, v) = arr ! lo
    seek tgt = go $ gallop ((tgt <=) . fst . (arr !)) (lo + 1) hi

-- galloping search: exponential probing followed by binary search.
-- O(log i) where i is the returned index.
gallop :: (Int -> Bool) -> Int -> Int -> Int
gallop p lo hi | lo >= hi  = hi
               | p lo      = lo
               | otherwise = go lo 1 where
  go lo step | lo + step >= hi = bisect lo hi
             | p (lo + step)   = bisect lo (lo + step)
             | otherwise       = go (lo + step) (step * 2)
  bisect l r | r - l <= 1 = r
             | p mid      = bisect l mid
             | otherwise  = bisect mid r
    where mid = (l+r) `div` 2

printTime :: String -> a -> IO ()
printTime label result = do
  putStr (label ++ ": "); hFlush stdout
  start_ps <- getCPUTime --in picoseconds
  end_ps <- result `seq` getCPUTime
  printf "%0.6fs\n" (fromIntegral (end_ps - start_ps) / (10^12))

{-# NOINLINE time2 #-}
time2 label xs ys = printTime label $ length $ toSorted $ xs `intersect` ys
{-# NOINLINE time3L #-}
time3L label xs ys zs = printTime label $ length $ toSorted $ (xs `intersect` ys) `intersect` zs
{-# NOINLINE time3R #-}
time3R label xs ys zs = printTime label $ length $ toSorted $ xs `intersect` (ys `intersect` zs)

thrice x = do x; x; x
main = do thrice $ time2 "odds & ends" odds ends
          thrice $ time2 "evens & odds" evens odds
          thrice $ time3L "(evens & odds) & ends" evens odds ends
          thrice $ time3R "evens & (odds & ends)" evens odds ends
```

Fig. 1. Benchmarking intersection for **Iter**

```
fromSortedArray :: Ord k => [(k,v)] -> Seek k v
fromSortedArray l = go 0 where
  hi = length l
  arr = listArray (0, hi) l
  go lo = Seek posn seek where
    posn | lo >= hi  = Bound Done
         | otherwise = Found k v where (k,v) = arr ! lo
    seek tgt = go $ gallop (satisfies tgt . fst . (arr !)) lo hi
```

Fig. 2. fromSortedArray for **Seek**

These nested seekable iterators also support disjunction/union, similar to the merge step in mergesort. We believe it is possible to extend this to existential quantification, essentially a disjunction over an entire trie level, using either a priority queue or a tournament tree [Knuth 1998, Chapter 5.4.1] to merge the sub-iterators. Moreover, by changing the lowest level of the trie from the unit type `()` to some other type, e.g. `Seek a (Seek b (Seek c Int))`, nested seekable iterators support *weighted logic programming*, where tuples of a relation have an associated value. Conventionally these values are drawn from a semiring: conjunction multiplies, while disjunction and existentials sum.

Together these amount to an implementation of tensor algebra. Indeed, our **Seek** interface is essentially equivalent to the *indexed streams* which Kovach et al. [2023] use as a tensor algebra compiler intermediate representation. Indexed streams replace the **Position** of an iterator with three values: its *index* (*key* in our terminology), a *ready flag* indicating whether it has found a value, and its *value* (which should only be accessed when the ready flag is set). In place of `seek` they have a *skip* function which takes a key and a flag indicating whether to use **Atleast** or **Greater** semantics.

To extend to full logic programming (weighted or not), however, we need *recursion,* which is hard to reconcile with the sorted nature of seekable iterators: self-reference means the existence of a larger key might imply the existence of a smaller one, so we may discover a key exists when it's too late to produce it. One approach might be to introduce an artifical time dimension, e.g. to represent a relation by a sequence of seekable iterators, representing a monotonically growing set, with recursion implemented as feedback with delay. This resembles a naïve implementation of Datalog; perhaps seminaïve evaluation is also possible.

References

Robert Atkey and Conor McBride. 2013. Productive Coprogramming with Guarded Recursion. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 197–208. doi:10.1145/2500365.2500597

Jon Louis Bentley and Andrew Chi-Chih Yao. 1976. An Almost Optimal Algorithm for Unbounded Searching. *Inf. Process. Lett.* 5, 3 (1976), 82–87. doi:10.1016/0020-0190(76)90071-5

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. *Log. Methods Comput. Sci.* 8, 4 (2012). doi:10.2168/LMCS-8(4:1)2012

Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. 2000. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, David B. Shmoys (Ed.). ACM/SIAM, 743–752. http://dl.acm.org/citation.cfm?id=338219.338634

Jason Hemann and Daniel P. Friedman. 2013. microKanren: A minimal functional core for relational programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme '13), Alexandria, VA, 2013*. University of Utah. http://webyrd.net/scheme-2013/papers/HemannMuKanren2013.pdf

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 192–203. doi:10.1145/1086365.1086390

Donald Ervin Knuth. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching* (2 ed.). Addison-Wesley. https://www.worldcat.org/oclc/312994415

Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1169–1193. doi:10.1145/3591268

M.Z. Kwiatkowska. 1989. Survey of fairness notions. *Information and Software Technology* 31, 7 (1989), 371–386. doi:10.1016/0950-5849(89)90159-6

Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 255–266. doi:10.1109/LICS.2000.855774

Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini (Eds.). ACM, 37–48. doi:10.1145/2213556.2213565

Susan S. Owicki and Leslie Lamport. 1982. Proving Liveness Properties of Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 455–495. doi:10.1145/357172.357178

Todd L. Veldhuizen. 2014. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 96–106. doi:10.5441/002/ICDT.2014.13

Hagen Völzer, Daniele Varacca, and Ekkart Kindler. 2005. Defining Fairness. In *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3653)*, Martín Abadi and Luca de Alfaro (Eds.). Springer, 458–472. doi:10.1007/11539452_35