Fair Intersection of Seekable Iterators

Michael Arntzenius
UC Berkeley

miniKanren 2025

RELATIONAL

LOGIC

disjunction / backtracking search

infinite

sparse

time to 1st solution

top-down

DATABASE

conjunction / intersecting search

finite

dense

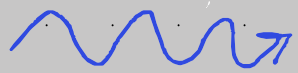
time to ALL solutions

bottom-up

EMBEDDED IN FP

μKanren

fair stream union



THIS TALK

"fair" iterator intersection

I · UNION

```
data Stream a = Nil | Cons a (Stream a)
```

LAZY!

```
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

INCOMPLETE!

```
primes = Cons 2 (Cons 3 (Cons 5 ...))

result = append primes (Cons 6 Nil)
```

$6 \notin result$ ∴ ☹

```
data Stream a = Nil | Cons a (Stream a)
```

LAZY!

```
interleave Nil ys              = ys
interleave (Cons x xs) ys = Cons x (interleave ys xs)
```

IS THIS COMPLETE?

```
data Stream a = Nil | Cons a (Stream a)
```

LAZY!

```
interleave Nil ys           = ys
interleave (Cons x xs) ys   = Cons x (interleave ys xs)
```

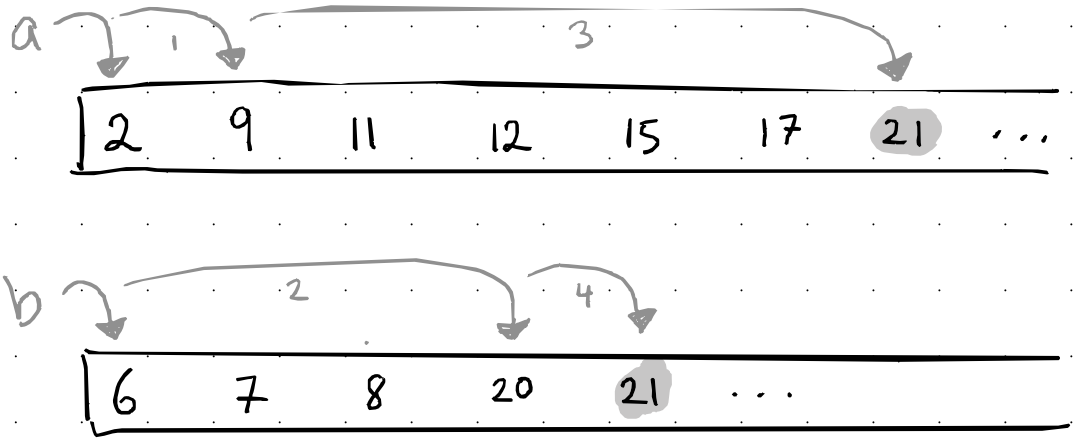
```
primes = Cons 2 (Cons 3 (Cons 5 (...)))
evenPrimes = filter even primes = Cons 2 ⊥
twoThreeFour = interleave evenPrimes (Cons 3 (Cons 4 Nil))
             = Cons 2 (Cons 3 ⊥)   where's 4 ?!
```

HOW DO WE AVOID ⊥ ?

```
data Stream a = Nil
              | Cons a (Stream a)
              | Later  (Stream a)

filter p Nil = Nil
filter p (Cons x xs) = if p x then Cons x xs' else xs'
    where xs' = filter p xs
filter p (Later xs)  = Later (filter p xs)

union Nil        ys = ys
union (Cons x xs) ys = Cons x (union xs ys)
union (Later xs) ys = Later (union ys xs)

primes = Cons 2 (Later (Cons 3 (Later (Cons 5 ...))))
twoThreeFour = union (filter even primes) (Cons 3 (Cons 4 Nil))
             = Cons 2 (Later (Cons 3 (Cons 4 (Later (Later (Lat
```

II · INTERSECTION

LEAPFROG INTERSECTION



STEP	a	b				
1	2 < 6	→	a.seek(≥6)	→	9	
2	9 > 6	→	b.seek(≥9)	→	20	
3	9 < 20	→	a.seek(≥20)	→	21	
4	21 > 20	→	b.seek(≥21)	→	21	→ YIELD!

IFACE: iter.seek(condition) → next elt satisfying condition

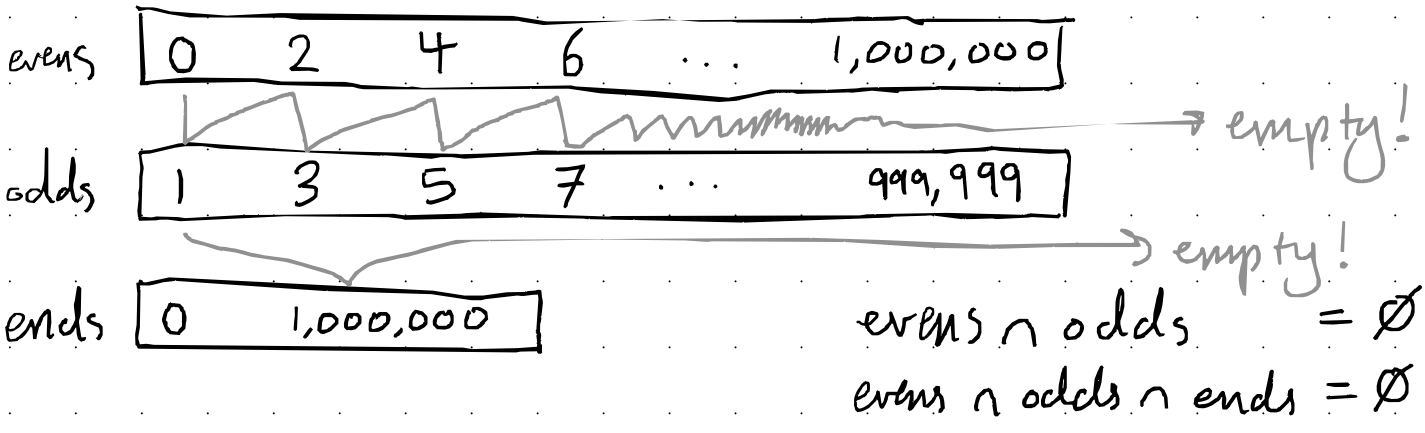
```
intersect(a, b).seek(cond):
  x = a.seek(cond)
  while true:
    y = b.seek(≥x)
    if x == y: break
    x = a.seek(≥y)
    if x == y: break
  return x
```

YIELD NEXT ELEMENT IN a ∩ b SATISFYING cond

COMPOSITIONAL INTERSECTION?

intersect : Iterator → Iterator → Iterator

eg. intersect(intersect(xs, ys), zs)



$evens \cap odds = \emptyset$

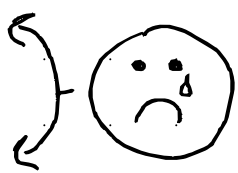
$evens \cap odds \cap ends = \emptyset$

`intersect(evens, odds).seek(≥0)` ✗ SLOOOW

`intersect(odds, ends).seek(≥0)` ✓ FAST

`intersect(intersect(evens, odds), ends)` ✗

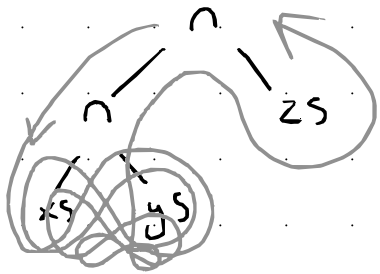
`intersect(evens, intersect(odds, ends))` ✓



NOT PERFORMANTLY ASSOCIATIVE!

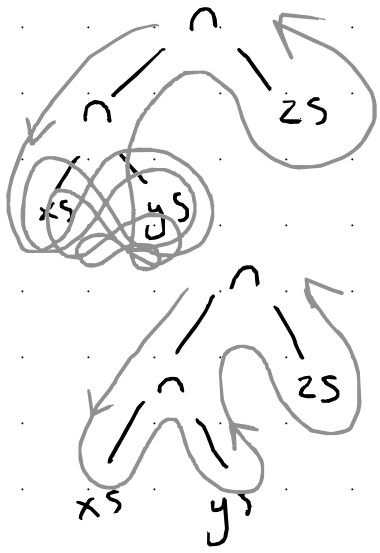
UNFAIR INTERFACE

iter.seek(cond) → next elt satisfying cond

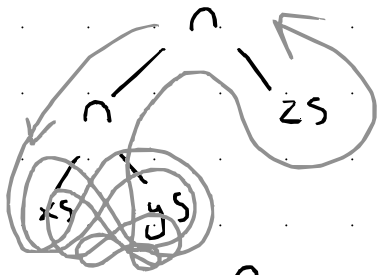


UNFAIR INTERFACE

iter.seek(cond) ⟶ next elt satisfying cond

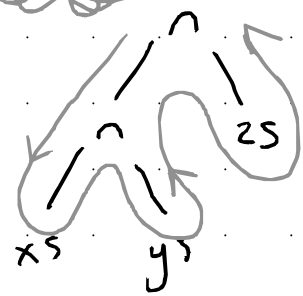


UNFAIR INTERFACE `iter.seek(cond)` ⟶ next elt satisfying cond



FAIR INTERFACE `iter.seek(cond)` ⟶ `(found, x)`

BOOL



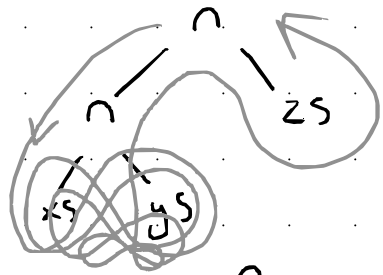
- x satisfies cond
- all remaining elements are $\geq x$
- found is true $\Longrightarrow$ x is yielded by the iterator (is in the sequence it represents)

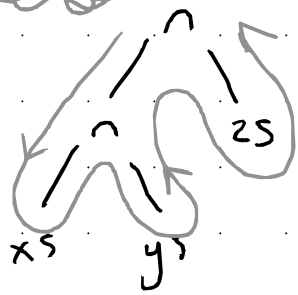
```
intersect(a, b).seek(cond):
    (xfound, x) = a.seek(cond)
    (yfound, y) = b.seek(≥x)
    found = xfound and yfound and x == y
    return (found, y)
```

UNFAIR INTERFACE `iter.seek(cond)` → next elt satisfying cond

FAIR INTERFACE `iter.seek(cond)` → `(found, x)`

BOOL






```
exhaust(iter):
    (found, x) = (false, -∞)
    while true:
        if found:
            yield x
            (found, x) = iter.seek(> x)
        else:
            (found, x) = iter.seek(≥ x)
```

RELATIONAL ALGEBRA

✓ NESTED INTERSECTION = RELATIONAL JOIN
= CONJUNCTION IN LOGIC PROG

foo(x,y) and bar(y,z)

⇓

```
for each x s.t  foo(x,_)
  for each y s.t  foo(x,y) and bar(y,_)
    for each z s.t  bar(y,z)
      yield (x,y,z)
```

CHUNKS of foo

INTERSECTION

"TRIE JOIN"

✓ UNIONS

foo(x) or bar(x)

like merge in merge-sort!

✓ EXISTENTIALS

$\exists x.\ foo(x, y)$

a "big" union across all x-values
— use a priority queue on y!

THE PROJECT

github.com/rntz/dijkstralog

GOAL efficient WEIGHTED (e.g. semiring) bottom-up logic programming, embedded in Rust

HAVE most of a runtime, no frontend

THE PAPER

https://www.rntz.net/files/on-fairness.pdf

SEE ALSO "Indexed Streams: A Formal IR for Fused Contraction Programs" Kovach et al PLDI '23