Generic Collections

One Interface to Rule Them All
What are “generic collections”? 
What is a collection?
What is a collection?

\[(1 \ 2 \ 3)\]
What is a collection?

Collections hold values.

(1 2 3)
Racket has lots of collections!

- Lists
- Vectors
- Hash Maps
- Queues
- Streams
- Growable Vectors
- Hash Sets
- Arrays
Q: How often do people actually use non-lists?
A: Not nearly enough.
Why?
But are they good enough?

**Lists**  
immutable

**Vectors**  
mutable by default

**Hash Maps**  
immutable by default

**Queues**  
always mutable

**Streams**  
immutable

**Growable Vectors**  
always mutable

**Hash Sets**  
immutable by default

**Arrays**  
immutable by default
But are they good enough?

- Lists: immutable
- Vectors: mutable by default
- Hash Maps: immutable by default
- Queues: always mutable
- Streams: immutable
- Growable Vectors: always mutable
- Hash Sets: immutable by default
- Arrays: immutable by default
It’s all about the APIs.

<table>
<thead>
<tr>
<th>Lists</th>
<th>Vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>vector-map</td>
</tr>
<tr>
<td>filter</td>
<td>vector-filter</td>
</tr>
<tr>
<td>reverse</td>
<td></td>
</tr>
<tr>
<td>for/list</td>
<td>for/vector</td>
</tr>
<tr>
<td>group-by</td>
<td></td>
</tr>
</tbody>
</table>
It’s all about the APIs.

<table>
<thead>
<tr>
<th>Lists</th>
<th>Vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>vector_map</td>
</tr>
<tr>
<td>filter</td>
<td>vector_filter</td>
</tr>
<tr>
<td>reverse</td>
<td></td>
</tr>
<tr>
<td>for/list</td>
<td>for/vector</td>
</tr>
<tr>
<td>group-by</td>
<td></td>
</tr>
</tbody>
</table>
Enter generic collections.
Generic collections provide a uniform interface to *all* collections while keeping the base set of primitives small.
Primitives

'(4 1 2 3)
> (conj #(1 2 3) 4)
'#(1 2 3 4)
> (conj (set 1 2 3) 4)
(set 1 3 2 4)
> (conj (set 1 2 3) 3)
(set 1 3 2)
> (set-nth '(1 2 3) 1 'b)
'(1 b 3)
> (set-nth #(1 2 3) 1 'b)
'#(1 b 3)
> (extend () '(1 2 3))
'#(1 2 3)
Primitives

> (first '(1 2 3))
1
> (rest '(1 2 3))
'(2 3)
> (first #(1 2 3))
1
> (rest #(1 2 3))
#<random-access-sequence>
> (nth '(1 2 3) 1)
2
> (nth #(1 2 3) 1)
2
> (random-access? '(1 2 3))
#f
> (random-access? #(1 2 3))
#t
> (require alexis/collection)
> (third #(1 2 3))
3
> (set-nth (stream 'a 'b 'c) 1 'B)
#<stream>
> (sequence->list (set-nth (stream 'a 'b 'c) 1 'B))
'(a B c)
> (apply + (set 1 1 2 3 5 8))
19
Immutability

```
> (first (vector 1 2 3))
first: contract violation
  expected: sequence?, which must be immutable
  given: '#(1 2 3), which is mutable
in: an and/c case of
  the 1st argument of
    (-> (and/c sequence? (not/c empty?)) any)
contract from:
  <pkgs>/alexis-collections/alexis/collection/collection.rkt
blaming: top-level
  (assuming the contract is correct)
at: <pkgs>/alexis-collections/alexis/collection/collection.rkt:44.3
```
What about the important functions?
fold

\[(\text{foldl} + 0 \ (1 \ 2 \ 3))\]
6

\[(\text{foldl} + 0 \ #\ (1 \ 2 \ 3))\]
6

\[(\text{foldl} + 0 \ (\text{set} \ 1 \ 2 \ 3))\]
6

\[(\text{foldl} + 0 \ (\text{stream} \ 1 \ 2 \ 3))\]
6

\[(\text{foldl} + 0 \ '(1 \ 2) \ #\ (3 \ 4) \ (\text{set} \ 5 \ 6) \ (\text{stream} \ 7 \ 8))\]
36
Q: What do we do for variadic functions that return collections?

(map + '(1 2 3) #(4 5 6))
A: Don’t return a concrete sequence... return a continuation!
This sounds a lot like lazy sequences.
map and filter

> (map + '(1 2 3) #(4 5 6))
#<stream>
> (filter even? (set 1 2 3 4 5 6 7))
#<stream>
map and filter

\[
> \text{(sequence->list (map + '(1 2 3) #(4 5 6)))}
\]
'(5 7 9)

\[
> \text{(sequence->list (filter even? (set 1 2 3 4 5 6 7)))}
\]
'(2 4 6)
Laziness is free!

> (define lazy-seq (map add1 (range 20)))
> lazy-seq
#<stream>
> (nth lazy-seq 15)
16
Laziness is fun (and useful)!

\[
\begin{align*}
> & \text{(define squares (map } \lambda (n) (* n n)) \text{ (naturals))} \\
> & \text{(nth squares 25)} \\
& 625 \\
> & \text{(define fibs (stream* 1 1 (map + fibs (rest fibs))))} \\
> & \text{(sequence->list (take 15 fibs))} \\
& '(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610) \\
> & \text{(define random-letters} \\
& \quad (\text{map integer->char (map (curry + 65) (randoms 26)))} \\
> & \text{(sequence->string (take 15 random-letters))} \\
& "\text{WEDWOHVSYILHTYN}"
\end{align*}
\]
Genericism can be more efficient!

\[
\begin{align*}
> & \ (\text{reverse } '(1\ 2\ 3\ 4)) \\
> & \ '(4\ 3\ 2\ 1) \\
> & \ (\text{reverse } #(1\ 2\ 3\ 4)) \\
> & \ #<\text{random-access-sequence}> \\
> & \ (\text{first } (\text{reverse } #(1\ 2\ 3\ 4))) \\
> & \ 4
\end{align*}
\]
Is this idiomatic Racket?
for loops

```plaintext
> (for ([x (take 5 (randoms))])
  (displayln x))

0.3295752491223747
0.4017543197993419
0.5215969193941353
0.27070311580464435
0.21192086885672548
```
for loops

> (define lazy
  (for/sequence ([(x i) (in-indexed (take 5 (randoms 10)))]
    (cons i x)))

> lazy
#<stream>

> (sequence->list lazy)
'(0 . 1) (1 . 7) (2 . 7) (3 . 4) (4 . 9)
for loops

> (define squares
  (for/sequence ([x (naturals)])
    (* x x)))

> squares
#<stream>

> (sequence->list (take 5 squares))
'(0 1 4 9 16)
Loops are extensible with for/sequence/derived.
match

> (match (stream 1 2 3 4)
   [(sequence a b c d) c])
3
> (match (stream 1 2 3 4)
   [(sequence a b ... c) b])
'(2 3)
> (match (naturals)
   [(sequence a b ...)
    (cons a b)])
'(0 . #<stream>)
Contracts

> (define/contract (sum seq)
  ((sequenceof number?) . -> . number?)
  (foldl + 0 seq))
> (sum (range 50))
1225
> (sum '(1 2 something-else 4))
sum: contract violation
  expected: number?
  given: 'something-else
  in: an element of
    the 1st argument of
      (-> (sequenceof number?) number?)
contract from: (function sum)
blaming: top-level
  (assuming the contract is correct)
at: eval:1.0
Demo
Thank you!

Packages

raco pkg install alexis-collections
raco pkg install alexis-pvector
raco pkg install alexis-collection-lens

GitHub

http://github.com/lexi-lambda