Functional Data Structures for Typed Racket

Hari Prashanth and Sam Tobin-Hochstadt
Northeastern University
Motivation

Typed Racket has very few data structures
Motivation

Typed Racket has very few data structures

Lists
Motivation

Typed Racket has very few data structures

Lists

Vectors
Motivation

Typed Racket has very few data structures

- Lists
- Vectors
- Hash Tables
Motivation

Typed Racket has very few data structures

Lists
Vectors
Hash Tables

Practical use of Typed Racket
Outline

- Motivation
- Typed Racket in a Nutshell
- Purely Functional Data Structures
- Benchmarks
- Typed Racket Evaluation
- Conclusion
Function definition in Racket

```racket
#lang racket

; Computes the length of a given list of elements
; length : list-of-elems -> natural
(define (length list)
  (if (null? list)
      0
      (add1 (length (cdr list)))))
```
#lang typed/racket

; Computes the length of a given list of integers
(: length : (Listof Integer) -> Natural)
(define (length list)
  (if (null? list)
      0
      (add1 (length (cdr list)))))
Function definition in Typed Racket

#lang typed/racket

; Computes the length of a given list of elements
(: length : (All (A) ((Listof A) -> Natural)))
(define (length list)
  (if (null? list)
      0
      (add1 (length (cdr list))))))
#lang racket

; Data definition of tree of integers

; A Tree is one of
; - null
; - BTree

(define-struct BTree
  (left
    (left
      elem
      right)))

; left and right are of type Tree
; elem is an Integer
Data definition in Typed Racket

#lang typed/racket

; Data definition of tree of integers

(define-type Tree (U Null BTree))

(define-struct: BTree
  ([left  : Tree]
   [elem  : Integer]
   [right : Tree]))
Data definition in Typed Racket

#lang typed/racket

; Polymorphic definition of Tree

(define-type (Tree A) (U Null (BTree A)))

(define-struct: (A) BTree
  ([left  : (Tree A)]
   [elem  : A]
   [right : (Tree A)]))
Outline

- Motivation
- Typed Racket in a Nutshell
- Purely Functional Data Structures
- Benchmarks
- Typed Racket Evaluation
- Conclusion
Destructive and Non-destructive update
Destructive and Non-destructive update

Destructive update
Destructive and Non-destructive update

Non-destructive update
(define-struct: (A) Queue
  ([front : (Listof A)]
   [rear : (Listof A)]))
Functional Queue

(define-struct: (A) Queue
  ([front : (Listof A)]
   [rear : (Listof A)]))
Functional Queue

(: dequeue : (All (A) ((Queue A) -> (Queue A))))
(define (dequeue que)
  (let ([front (cdr (Queue-front que))]
        [rear (Queue-rear que)])
    (if (null? front)
        (Queue (reverse rear) null)
        (Queue front rear))))
Functional Queue

Queue $q$

(for ([id (in-range 100)])
  (dequeue q))
Banker’s Queue [Okasaki 1998]

Lazy evaluation solves this problem
Banker’s Queue [Okasaki 1998]

Lazy evaluation solves this problem

(\(\text{val} : (\text{Promise Exact-Rational})\))

(\(\text{define val} (\text{delay} (/ 5 0))\))
Banker’s Queue [Okasaki 1998]

Lazy evaluation solves this problem

Streams

(define-type (Stream A)
   (Pair A (Promise (Stream A)))))
Banker’s Queue [Okasaki 1998]

Lazy evaluation solves this problem

(define-struct: (A) Queue
  ([front : (Stream A)]
   [lenf : Integer]
   [rear : (Stream A)]
   [lenr : Integer])))

Invariant lenf >= lenr
Banker’s Queue [Okasaki 1998]

Lazy evaluation solves this problem

(: check :
 (All (A) (Stream A) Integer (Stream A) Integer -> (Queue A)))

(define (check front lenf rear lenr)
 (if (>= lenf lenr)
     (if (make-Queue front lenf rear lenr)
         (make-Queue (stream-append front (stream-reverse rear))
                     (+ lenf lenr) null 0))))
Banker’s Queue [Okasaki 1998]

Lazy evaluation solves this problem

(make-Queue (stream-append front (stream-reverse rear))
(+ lenf lenr) null 0)
Banker’s Queue [Okasaki 1998]

Lazy evaluation solves this problem

Amortized running time of $O(1)$ for the operations

enqueue, dequeue and head
Real-Time Queues [Hood & Melville 81]
Real-Time Queues [Hood & Melville 81]

Eliminating amortization by Scheduling
Real-Time Queues [Hood & Melville 81]

Eliminating amortization by Scheduling

Banker’s Queue - reverse is a forced completely
Real-Time Queues [Hood & Melville 81]

Eliminating amortization by Scheduling

\[
(\text{rotate} : \\
(\text{All } (A) \ (\text{(Stream } A) \ (\text{Listof } A) \ (\text{Stream } A) \rightarrow (\text{Stream } A))))
\]

\[
(\text{define } (\text{rotate front rear accum}) \\
(\text{if } (\text{empty-stream? front}) \\
(\text{stream-cons } (\text{car rear) accum}) \\
(\text{stream-cons } (\text{stream-car front}) \\
(\text{rotate } (\text{stream-cdr front}) \\
(\text{cdr rear}) \\
(\text{stream-cons } (\text{car rear) accum}))))))
\]

Incremental reversing
Real-Time Queues [Hood & Melville 81]

Eliminating amortization by Scheduling

Worst-case running time of $O(1)$ for the operations

enqueue, dequeue and head
Binary Random Access Lists [Okasaki 1998]

Nat is one of
- 0
- (add1 Nat)

List is one of
- null
- (cons elem List)
Binary Random Access Lists [Okasaki 1998]

Nat is one of
- 0
- (add1 Nat)

List is one of
- null
- (cons elem List)

**cons** corresponds to increment
**cdr** corresponds to decrement
**append** corresponds to addition
Binary Random Access Lists [Okasaki 1998]

(define-type (RAList A) (Listof (Digit A)))
Binary Random Access Lists [Okasaki 1998]

```
(define-type (RAList A) (Listof (Digit A)))
(define-type (Digit A) (U Zero (One A)))
```
Binary Random Access Lists [Okasaki 1998]

(define-struct: Zero () )
Binary Random Access Lists [Okasaki 1998]

(define-struct: Zero ()

(define-struct: (A) One ([fst : (Tree A)]))
Binary Random Access Lists [Okasaki 1998]

(define-type (Tree A) (U (Leaf A) (Node A))))
Binary Random Access Lists [Okasaki 1998]

```
(define-type (Tree A) (U (Leaf A) (Node A)))
(define-struct: (A) Leaf ([fst : A]))
```
Binary Random Access Lists [Okasaki 1998]

(define-type (Tree A) (U (Leaf A) (Node A)))

(define-struct: (A) Leaf ([fst : A]))

(define-struct: (A) Node
  ([size   : Integer]
   [left   : (Tree A)]
   [right  : (Tree A)]))
Binary Random Access Lists [Okasaki 1998]

(define-type (RAList A) (Listof (Digit A)))

0

[  ]

(list)
Binary Random Access Lists [Okasaki 1998]

(define-type (RAList A) (Listof (Digit A)))

\[
\begin{bmatrix}
1 & \text{\textbf{3}}
\end{bmatrix}
\]

(list 3)
Binary Random Access Lists [Okasaki 1998]

(define-type (RAList A) (Listof (Digit A)))

```
10
[
  [8 3]
(8 3)
```

45
Binary Random Access Lists [Okasaki 1998]

(define-type (RAList A) (Listof (Digit A)))
Binary Random Access Lists [Okasaki 1998]

(define-type (RAList A) (Listof (Digit A)))

100

[ [ [ 1 7 8 3 ] ] ]

(list 1 7 8 3)
Binary Random Access Lists [Okasaki 1998]

\[
\text{(define-type (RAList A) (Listof (Digit A)))}
\]

\[
\begin{array}{c}
101 \\
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\left[
\begin{array}{c}
4 \\
1 \\
7 \\
8 \\
3
\end{array}
\right]
\end{array}
\end{array}
\]

\[
\text{(list 4 1 7 8 3)}
\]
Binary Random Access Lists [Okasaki 1998]

(define-type (RAList A) (Listof (Digit A)))

110

(list 12 4 1 7 8 3)
Binary Random Access Lists [Okasaki 1998]

\[(\text{define-type~(RAList~A)~(Listof~(Digit~A))})\]

Worst-case running time of $O(\log n)$ for the operations \text{cons, car, cdr, lookup} and \text{update}
VLists [Bagwell 2002]

(define-struct: (A) Base
  ([previous : (U Null (Base A))]
   [elems : (RAList A)]))

(define-struct: (A) VList
  ([offset : Natural]
   [base : (Base A)]
   [size : Natural])))
VLists [Bagwell 2002]

List with one element - 6

```
```

Base

Offset

6
VLists [Bagwell 2002]

**cons** 5 and 4 to the previous list
**VLists [Bagwell 2002]**

`cons` 3 and 2 to the previous list
VLists [Bagwell 2002]

cdr of the previous list
VLLists [Bagwell 2002]

Random access takes $O(1)$ average and $O(\log n)$ in worst-case.
Our library

Library has 30 data structures which include

- Variants of Queues
- Variants of Deques
- Variants of Heaps
- Variants of Lists
- Red-Black Trees
- Tries
- Sets
- Hash Lists
Our library

Library has 30 data structures
Our library

Library has 30 data structures

Data structures have several utility functions
Our library

Library has 30 data structures

Data structures have several utility functions

Our implementations follow the original work
Outline

- Motivation
- Typed Racket in a Nutshell
- Purely Functional Data Structures
- Benchmarks
- Typed Racket Evaluation
- Conclusion
Benchmarks

Queue benchmarks - enqueue

Time in ms

Number of elements

1000 10000 100000 1000000

Physicist's Banker's Bootstrapped Imperative

**Benchmarking done with 2.1 GHz Intel Core 2 Duo (Linux) machine using Racket version 5.0.0.9

(foldl enqueue que list-of-100000-elems)
Benchmarks

Queue benchmarks - dequeue

**Benchmarks done with 2.1 GHz Intel Core 2 Duo (Linux) machine using Racket version 5.0.0.9**
Benchmarks

Heap benchmarks - insert

Time in ms

Number of elements

- Binomial
- Pairing
- Bootstrapped
- Imperative

** Benchmarking done with 2.1 GHz Intel Core 2 Duo (Linux) machine using Racket version 5.0.0.9**
Benchmarks

Heap benchmarks - find-min/max

- **Binomial**
- **Pairing**
- **Bootstrapped**
- **Imperative**

**Time in ms**

**Number of elements**

1000, 10000, 100000, 1000000

**Benchmarking done with 2.1 GHz Intel Core 2 Duo (Linux) machine using Racket version 5.0.0.9**
Benchmarks

Heap benchmarks - delete-min/max

- Binominal
- Pairing
- Bootstrapped
- Imperative

Time in ms

Number of elements

100000
1000000

**Benchmarking done with 2.1 GHz Intel Core 2 Duo (Linux) machine using Racket version 5.0.0.9**
Outline

- Motivation
- Typed Racket in a Nutshell
- Purely Functional Data Structures
- Benchmarks
- Typed Racket Evaluation
- Conclusion
ML to Typed Racket

ML idioms can be easily ported to Typed Racket
ML to Typed Racket

ML idioms can be easily ported to Typed Racket

```racket
(type 'a Queue = int * 'a Stream * int * 'a Stream)

(define-struct: (A) Queue
  ([lenf : Integer]
   [front : (Stream A)]
   [lenr : Integer]
   [rear : (Stream A)]))
```
ML to Typed Racket

ML idioms can be easily ported to Typed Racket

type 'a Queue = 'a list * int * 'a list susp * int * 'a list

(define-struct: (A) Queue
  ([pref : (Listof A)]
   [lenf : Integer]
   [front : (Promise (Listof A))]
   [lenr : Integer]
   [rear : (Listof A)]))
Optimizer in Typed Racket

Optimizer based on type information
Optimizer in Typed Racket

Optimizer Benchmarks

- **Binomial Heap**: 5470 ms (With Optimizer), 6535 ms (Without Optimizer)
- **VList**: 10140 ms (With Optimizer), 11065 ms (Without Optimizer)
- **Leftist Heap**: 19790 ms (With Optimizer), 26530 ms (Without Optimizer)
- **Banker's Queue**: 14260 ms (With Optimizer), 22800 ms (Without Optimizer)

Time in ms

Data structures
Polymorphic recursion

\[
\text{(define-type (Seq A) (Pair A (Seq (Pair A A))))}
\]

Non-uniform type
Polymorphic recursion

(define-type (EP A) (U A (Pair (EP A) (EP A)))))
(define-type (Seq A) (Pair (EP A) (Seq A)))

Uniform type
Conclusion

Typed Racket is useful for real-world software.

Functional data structures in Typed Racket are useful and performant.

A comprehensive library of data structures is now available.
Thank you...

Library is available for download from

http://planet.racket-lang.org/