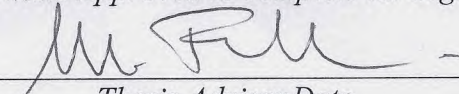


**NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
MS THESIS APPROVAL FORM**

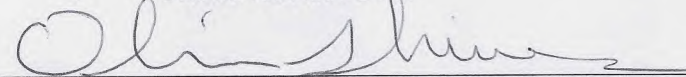
THESIS TITLE: FUNCTIONAL DATA STRUCTURES FOR TYPED RACKET

AUTHOR: HARI PRASANTH K R

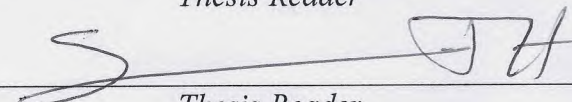
^{MS}
~~PH.D.~~ Thesis Approved to complete all degree requirements for the ^{MS}~~Ph.D.~~ Degree in Computer Science.


Thesis Advisor Date

22 Feb 2011


Thesis Reader

2011/2/21
Date

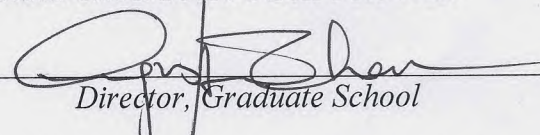

Thesis Reader

2011-2-22
Date

Thesis Reader

Date

GRADUATE SCHOOL APPROVAL:


Director, Graduate School

2011-2-23
Date

ORIGINAL COPY DEPOSITED IN LIBRARY:

Reference Librarian

Date

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature

Date

Distribution: Original to Reference Library (Boxed- with signature approval sheet)
One Copy to Thesis Advisor
One Copy to Each Member of Thesis Committee
One Copy to Director of Graduate School
One Copy to Graduate School Office (with signature approval sheet, including all signatures)

Copies submitted to those other than Reference Library should be bound in a black hardcover binder with a squeeze back. (Available in N.U. Bookstore).

Functional Data Structures for Typed Racket

A Master's thesis presented
by

Hari Prashanth K R

Thesis Advisor
Prof. Matthias Felleisen

Readers
Dr. Olin Shivers

Dr. Sam Tobin-Hochstadt

February 22, 2011
College of Computer and Information Science
Northeastern University
440 Huntington Avenue
Boston, Massachusetts 02115

UMI Number: 1491239

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 1491239

Copyright 2011 by ProQuest LLC.

All rights reserved. This edition of the work is protected against unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

In the past two decades, a wide variety of functional data structures have been developed and proved to be practical. These data structures appear in functional languages such as Haskell, OCaml and Clojure. Although Typed Racket provides some valuable functional data structures, these are insufficient to fully support functional programming. This MS thesis presents a library of functional data structures for Typed Racket and reports on their feasibility and performance. At the same time, it presents an evaluation of Typed Racket's suitability for developing large libraries.

Acknowledgments

I express my sincere gratitude to Prof. Matthias Felleisen for introducing me to the exciting world of functional programming, for his continuous support and motivation. I could not have imagined having a better teacher and advisor.

Besides my advisor, I would like to thank Dr. Sam Tobin-Hochstadt for his patience, enthusiasm and constant encouragement. His guidance helped me in all the time of research and writing of this thesis.

Thanks also to Dr. Olin Shivers for taking time off from his busy schedule to be part of my thesis committee. I am grateful to Dr. Eli Barzilay for helping me refine and polish this work. Without his help this thesis would not have been the same.

I thank my fellow lab mates and friends in Northeastern Programming Research Lab for many stimulating discussions.

Last but not the least, I would like to thank my family: my parents and my brothers for their love and encouragement.

Contents

1	Introduction	1
2	An Overview to Typed Racket	3
2.1	Data Definitions	3
2.2	Function Definitions	4
2.3	Polymorphic Definitions	5
2.4	Integration of typed and untyped code	6
2.5	A Simple Queue	7
3	Functional Data Structures	9
3.1	Purely Functional Data Structures	9
3.2	Choice of Data Structures	10
3.3	Data Structure Interfaces	11
3.3.1	Queue Interface	11
3.3.2	Interface for Deque	12
3.3.3	Heap Interface	12
3.3.4	List Interface	13
3.4	Data Structure Implementation	15
3.4.1	Queue Implementations	15
3.4.2	Deque Implementations	18
3.4.3	Heap Implementations	19
3.4.4	List Implementations	23
3.4.5	Other Data Structures	25
4	Performance Analysis	29
4.1	Queue Performance	29
4.2	Heap Performance	33
4.3	List Performance	36
4.4	Conclusion	37

5	Evaluation of Typed Racket	39
5.1	Benefits of Typed Racket	39
5.2	Disadvantages of Typed Racket	42
6	Related Work	47
6.1	In Other Languages	47
6.2	In Racket	47
7	Conclusion	49
7.1	Contributions	49
7.2	Future Work	49
	Bibliography	51

Introduction

The past decade has seen advancements in the development of efficient functional data structures, particularly by Okasaki (1998a) and Bagwell (2002, 2000). However, few attempts have been made to port these structures to Racket¹.

In this MS thesis, I present a comprehensive library of efficient functional data structures in Typed Racket (Tobin-Hochstadt and Felleisen, 2008; Tobin-Hochstadt, 2010), a recently developed typed dialect of Racket. The thesis also includes an investigation of the usefulness of these data structures and evaluation of the practicality of Typed Racket in the context of functional data structures.

The programming productivity in a language is largely determined by the data structures that are built-in to the language. Data structures available in C-like languages cannot be easily adapted to functional languages like Typed Racket as they depend crucially on mutation, which is discouraged and/or disallowed in functional languages. So, functional language require functional data structures to support functional programming.

In response, I have developed the library of functional data structures for Typed Racket

- Variants of FIFO queues including those discussed by Okasaki (1998a) and Hood-Melville Queues (Hood and Melville, 1980).
- Variants of priority queues or heaps which include Binomial Heaps, Pair-

¹<http://planet.plt-scheme.org/display.ss?package=ralist.plt&owner=dvanhorn>
<http://planet.plt-scheme.org/display.ss?package=galore.plt&owner=soegaard>

ing Heaps (Fredman et al., 1986), Leftist Heap (Crane, 1972), Bootstrapped Heaps (Okasaki, 1998a) and Skew Binomial Heap.

- Variants of lists such as random access lists Okasaki (1998a), catenable lists (Okasaki, 1998a) and vlists (Bagwell, 2002)
- Other data structures such as Red-Black Trees Okasaki (1999), Treaps (Aragon and Seidel, 1989), Hash Lists (Bagwell, 2002), Tries (Bagwell, 2000) and Sets.

Historically, Racket and Typed Racket provide three data structures: linked lists, which supports many forms of functional programming, vectors, and hash tables. However, these are not sufficient. To truly support efficient programming in a functional style, additional functional data structure support is required.

Typed Racket has a novel type checker, based on *occurrence typing*, that ensures type safety, supports all the Racket idioms and inter-operates well with Racket. Even though Typed Racket has seen use in the past several years, all of this use is in classroom settings. The development of a robust functional data structure library in Typed Racket, thus provides an opportunity to evaluate the usefulness and practicality of Typed Racket.

An Overview to Typed Racket

Typed Racket (Tobin-Hochstadt and Felleisen, 2008; Tobin-Hochstadt, 2010) is a statically typed sister language of Racket. The purpose of Typed Racket is to enable the gradual and straightforward migration from the untyped to the typed language. This chapter gives an overview of the features of Typed Racket that are used in later chapters.

2.1 Data Definitions

A Data Definition states, in a mixture of English and Racket, how a class of structures are used and how an element of the class of data is constructed. Data Definition 1 shows an example.

```
#lang racket
```

Data Definition 1

```
;; Data definition for a simple queue of integers
```

```
(struct IntQueue  
  (front rear))
```

```
;; front and rear are list-of integers.
```

The structure *IntQueue* in Data Definition 1 has two fields *front* and *rear*. As part of the data definition, both the fields of *IntQueue* structure are declared to be list of integers in the comment. But no guarantees are made by the language about the types that are stated in the comment.

In contrast, in Typed Racket the fields of the *IntQueue* structure must be annotated with types and the type checker makes sure that the fields have the

right type and no extra comments are required here. Data Definition 2 the definition of *IntQueue* in Typed Racket.

```
#lang typed/racket
```

Data Definition 2

```
;; Data definition for a simple queue of integers  
(struct: IntQueue  
  ([front : (Listof Integer)]  
   [rear  : (Listof Integer)]))
```

struct: is used to define structures in Typed Racket. And both the fields have been annotated to be of type **(Listof Integer)**.

Typed Racket provides **define-type** to define new types. Data Definition 3 defines the type *IntString* which is the union of **Integer** and **String**. Here, **U** is used for the union of two or more types.

```
#lang typed/racket
```

Data Definition 3

```
(define-type IntString (U Integer String))
```

2.2 Function Definitions

The functions in this section are defined using the data definitions from Section 2.1. The functions in Racket use the Racket definition of *IntQueue* and Typed Racket definitions use the Typed Racket definition of *IntQueue*.

Function Definition 1 shows the definition of the function *head* in Racket. *head* returns the first element from the given *IntQueue*.

```
#lang racket
```

Function Definition 1

```
;; head : IntQueue → Integer  
;; Returns the first element of the given queue  
(define (head queue)  
  (if (null? (IntQueue-front queue))  
      (error 'head "given queue is empty")  
      (first (IntQueue-front queue))))
```

Racket's convention of writing the input and output types of the function as comments change to actual types in Typed Racket. The infix identifier \rightarrow is used to specify function types. Function Definition 2 shows the definition of the function *head* in Typed Racket.

```
#lang typed/racket
```

Function Definition 2

```
(: head : IntQueue  $\rightarrow$  Integer)  
;; Returns the first element of the given queue  
(define (head queue)  
  (if (null? (IntQueue-front queue))  
      (error 'head "given queue is empty")  
      (first (IntQueue-front queue))))
```

2.3 Polymorphic Definitions

Typed Racket supports polymorphism. For example, Data Definition 4 shows the polymorphic definition of queue using two lists in Typed Racket.

```
#lang typed/racket
```

Data Definition 4

```
;; Data definition for a simple queue  
(struct: ( $\alpha$ ) Queue  
  ([front : (Listof  $\alpha$ )]  
   [rear : (Listof  $\alpha$ )]))
```

In Data Definition 4, α is a type variable. The fields *front* and *rear* are of type `(Listof α)`.

The function *head* on the polymorphic structure **Queue** is defined in Function Definition 3.

```
#lang typed/racket
```

Function Definition 3

```
(: head : (∀ (α) (Queue α) → α))
;; Returns the first element of the given queue
(define (head queue)
  (if (null? (Queue-front queue))
      (error 'head "given queue is empty")
      (first (Queue-front queue))))
```

The Polymorphic type constructor \forall is used to construct the type for polymorphic functions. Again, α is the polymorphic type variable.

2.4 Integration of typed and untyped code

Typed Racket code smoothly interoperates with Racket code. Untyped module can imported into typed modules. For example, typed module *M2* imports untyped module *M1*.

```
#lang racket
;; Data definition for a simple queue of integers
(struct IntQueue
  (front rear))

(provide (struct-out IntQueue))
```

M1

```
#lang typed/racket
(require/typed M1 [struct IntQueue ([front : (Listof Integer)]
                                     [rear : (Listof Integer)])])

(: head : IntQueue → Integer)
;; Returns the first element of the queue
(define (head queue)
  (if (null? (IntQueue-front queue))
      (error 'head "given queue is empty")
      (first (IntQueue-front queue))))

(provide head (struct-out IntQueue))
```

M2

The **require/typed** form used in module *M2* specifies the types for imported untyped code. Module *M2* imports the untyped structure *IntQueue* from module *M1* and specifies the types for its fields.

Importing typed modules into untyped modules is also straightforward. The untyped module *M3* imports the typed module *M2* uses the function *head*.

```
#lang racket  
(require M2)
```

M3

```
(head (make-IntQueue (list 1 2 3) (list 6 5 4)))  
(head (make-IntQueue (list) (list)))
```

2.5 A Simple Queue

A Batched Queue is a simple implementation of a Queue data structure using two lists. Figure 2.1 shows a Typed Racket implementation of Batched Queue. The implementation includes the the basic operations on the Queue structure namely, *empty?*, *head*, *dequeue* and *enqueue*.


```
#lang typed/racket
```

Data Structure 1

```
(struct: ( $\alpha$ ) Queue
  ([front : (Listof  $\alpha$ )]
   [rear : (Listof  $\alpha$ )]))

(: empty? : ( $\forall$  ( $\alpha$ ) (Queue  $\alpha$ )  $\rightarrow$  Boolean))
;; Returns true if the queue is empty
(define (empty? queue)
  (null? (Queue-front queue)))

(: balance : ( $\forall$  ( $\alpha$ ) (Listof  $\alpha$ ) (Listof  $\alpha$ )  $\rightarrow$  (Queue  $\alpha$ )))
(define (balance front rear)
  (if (null? front)
    (Queue (reverse rear) null)
    (Queue front rear)))

(: head : ( $\forall$  ( $\alpha$ ) (Queue  $\alpha$ )  $\rightarrow$   $\alpha$ ))
;; Returns the first element of the queue
(define (head queue)
  (if (empty? queue)
    (error 'head "given queue is empty")
    (first (Queue-front queue))))

(: dequeue : ( $\forall$  ( $\alpha$ ) (Queue  $\alpha$ )  $\rightarrow$  (Queue  $\alpha$ )))
;; Returns the rest of the queue
(define (dequeue queue)
  (if (empty? queue)
    (error 'dequeue "given queue is empty")
    (balance (rest (Queue-front queue)) (Queue-rear queue))))

(: enqueue : ( $\forall$  ( $\alpha$ )  $\alpha$  (Queue  $\alpha$ )  $\rightarrow$  (Queue  $\alpha$ )))
;; Inserts the given element into the queue
(define (enqueue elem queue)
  (Queue (Queue-front queue)
    (cons elem (Queue-rear queue))))
```

Figure 2.1: Typed Racket implementation of a Batched Queue

Functional Data Structures

A data structure is purely functional if it supports only non-destructive updates. These data structures are persistent, which is especially advantageous when dealing with multiple versions of the same object. This chapter describes the functional data structure implemented for this thesis.

3.1 Purely Functional Data Structures

In imperative data structures, only one version of the data structure is available. An update on an imperative data structure destroys the previous version. Such updates are therefore known as destructive updates. In contrast, functional data structures support only nondestructive updates, which preserves both the old and new version of the data structure for further processing. The data structures that preserve their previous version when they are modified are said to be persistent. Figure 3.2 and Figure 3.3 respectively demonstrate destructive and nondestructive update on the binary tree from Figure 3.1.

The property of persistence makes purely functional data structures the right choice for the applications that need to maintain multiple versions of the data structure especially to support programming in functional style and immutability in these data structures makes it easier to reason about the parallelism in programs. Operations on purely functional data structure can be performed in parallel because there are no side effects changing the data structure.

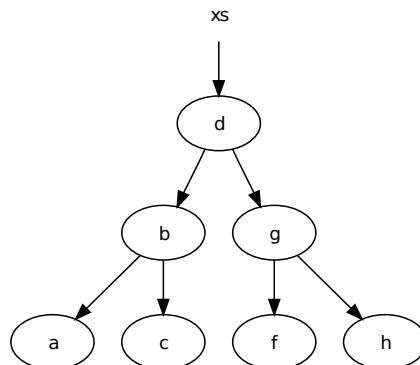


Figure 3.1: A Binary Tree

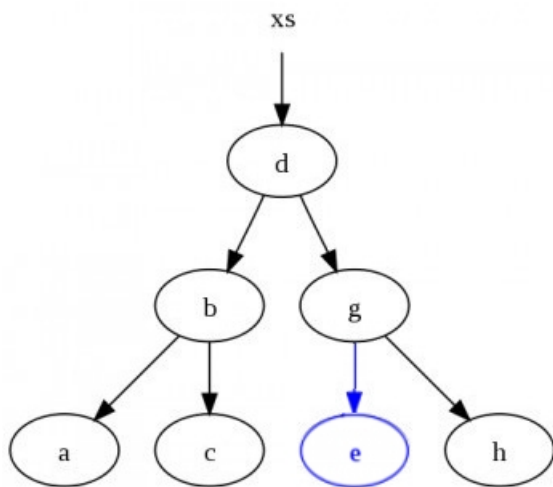


Figure 3.2: Destructive Update

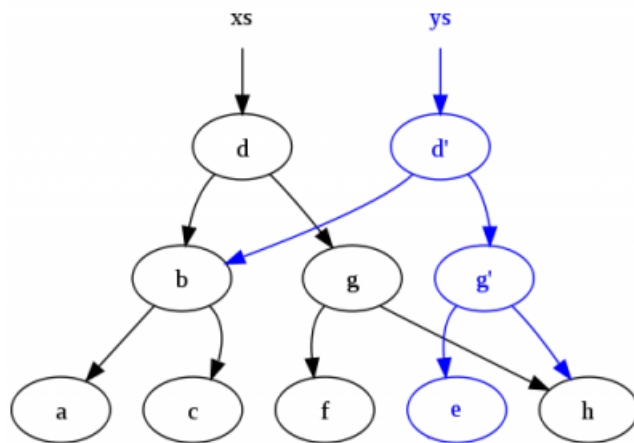


Figure 3.3: Nondestructive Update

3.2 Choice of Data Structures

The chosen data structures provide a wide variety of performance characteristics and APIs. They include data structures designed for particular use cases and those that are only performant for certain operations. These include a variety of single and double-ended queues (otherwise known as deques), variants of heaps and sets, red-black trees, treaps, tries and hash lists. Implementing a wide variety of data structures with the same API allows studying the practical performance characteristics of each variant, since different variants provide

widely varying running times for specific operations.

Another factor influencing the variety of data structures is Typed Racket. By choosing a wide variety of data structures, each with its own type invariants and type system requirements, Typed Racket's type system is exercised and validated in many different ways.

3.3 Data Structure Interfaces

All the data structures in the library are polymorphic in their element type and provide utility functions such as *map*, *fold* and *filter* as well as basic API operations. Each of the queue implementations share a common API, and similarly for dequeues, heaps and other data structures. The following subsections describe the basic interface of the each category of data structure.

3.3.1 Queue Interface

Queues are First-In-First-Out (FIFO) data structures. All the queue implementations in the library use the type **(Queue α)**, where α is the polymorphic type variable. Each queue data structure shares the following common interface:

```
;; Constructs a queue of type (Queue  $\alpha$ ).
```

```
(: queue : ( $\forall$  ( $\alpha$ )  $\alpha$  *  $\rightarrow$  (Queue  $\alpha$ )))
```

```
;; Checks if the given queue is empty
```

```
(: empty? : ( $\forall$  ( $\alpha$ ) (Queue  $\alpha$ )  $\rightarrow$  Boolean))
```

```
;; Returns the first element of the given queue.
```

```
(: head : ( $\forall$  ( $\alpha$ ) (Queue  $\alpha$ )  $\rightarrow$   $\alpha$ ))
```

```
;; Returns the tail of the given queue, i.e. a queue without the  
;; first element of the given queue.
```

```
(: tail : ( $\forall$  ( $\alpha$ ) (Queue  $\alpha$ )  $\rightarrow$  (Queue  $\alpha$ )))
```

```
;; Inserts the given element into the given queue.
```

```
(: enqueue : ( $\forall$  ( $\alpha$ )  $\alpha$  (Queue  $\alpha$ )  $\rightarrow$  (Queue  $\alpha$ )))
```

Queue Interface

3.3.2 Interface for Deque

Double ended queues are also known as dequeues. Elements can be inserted and removed from either end of a deque. The following basic interface are provided by deque data structure and they have the type **(Deque α)**:

<pre>;; Constructs a deque of type (Deque α). (: deque : (\forall (α) α * \rightarrow (Deque α))) ;; Checks if the given deque is empty (: empty? : (\forall (α) ((Deque α) \rightarrow Boolean))) ;; Returns the first element of the given deque. (: head : (\forall (α) (Deque α) \rightarrow α)) ;; Returns the last element of the given deque. (: last : (\forall (α) (Deque α) \rightarrow α)) ;; Returns the tail of the given deque, i.e. a deque without the first ;; element of the given deque. (: tail : (\forall (α) (Deque α) \rightarrow (Deque α))) ;; Returns a deque without the last element of the given deque. (: init : (\forall (α) (Deque α) \rightarrow (Deque α))) ;; Inserts the given element to the rear of the given deque. (: enqueue : (\forall (α) α (Deque α) \rightarrow (Deque α))) ;; Inserts the given element to the front of the given deque. (: enqueue-front : (\forall (α) α (Deque α) \rightarrow (Deque α)))</pre>	Deque Interface
---	-----------------

3.3.3 Heap Interface

Heaps are tree-based data structures that satisfy two additional constraints:

Shape All its levels of the tree must be full except the last level where only rightmost leaves may be missing.

Parental Dominance The key at each node of the tree must greater than or equal (max-heap) OR less than or equal (min-heap) to the keys of its children.

All the heaps have the type **(Heap α)**:

<pre>;; Constructs a heap with the given elements and ;; ($\alpha \rightarrow \mathbf{Boolean}$) as its comparison function. (: heap : ($\forall (\alpha) (\alpha \rightarrow \mathbf{Boolean}) \alpha * \rightarrow (\mathbf{Heap} \alpha)$)) ;; Checks if the given heap is empty. (: empty? : ($\forall (\mathbf{A}) ((\mathbf{Heap} \mathbf{A}) \rightarrow \mathbf{Boolean})$)) ;; Inserts the given element into the given heap. (: insert : ($\forall (\alpha) \alpha (\mathbf{Heap} \alpha) \rightarrow (\mathbf{Heap} \alpha)$)) ;; Returns the min or the max element of the given heap. (: find-min/max : ($\forall (\alpha) (\mathbf{Heap} \alpha) \rightarrow \alpha$)) ;; Deletes the min or the max element of the given heap. (: delete-min/max : ($\forall (\alpha) (\mathbf{Heap} \alpha) \rightarrow (\mathbf{Heap} \alpha)$))</pre>	<div style="border: 1px solid black; padding: 2px 5px; display: inline-block;">Heap Interface</div>
---	---

3.3.4 List Interface

Alternatives to the Racket's built-in lists. They have the type **(List α)**¹ and provide the following functions:

¹Typed Racket also provides a built-in type **(List α)**. The two types can be disambiguated using Racket's module system.

```
;; Constructs a list with the given elements.
```

```
(: list : (∀ (α) α * → (List α)))
```

List Interface

```
;; Checks if the given list is empty.
```

```
(: empty? : (∀ (α) ((List α) → Boolean)))
```

```
;; Adds the given element to the front of the given list.
```

```
(: cons : (∀ (α) α (List α) → (List α)))
```

```
;; Returns the first element of the given list.
```

```
(: first : (∀ (α) (List α) → α))
```

```
;; Returns the list without the first element of the given list.
```

```
(: rest : (∀ (α) (List α) → (List α)))
```

Random Access List

Random Access Lists are lists with efficient array-like random access operations. These include *list-ref* and *list-set* (a functional analogue of *vector-set!*). Random Access Lists extend the basic list interface with the following operations:

```
;; Returns the element at a given location in the list.
```

```
(: list-ref : (∀ (α) Natural (List α) → (List α)))
```

RAList Interface

```
;; Updates the element at a given location in the list with a new element.
```

```
(: list-set : (∀ (α) Natural (List α) α → (List α)))
```

Catenable List

Catenable Lists are a list data structure with an efficient append operation:

```
;; Appends several lists together.
```

```
(: append : (∀ (α) (List α) * → (List α)))
```

Catenable List Interface

```
;; Inserts a given element to the rear end of the list.
```

```
(: cons-to-end : (∀ (α) α (List α) → (List α)))
```

3.4 Data Structure Implementation

This section describes the theory and performance characteristics of each data structure in the library.

3.4.1 Queue Implementations

Banker's Queue

Banker's Queues (Okasaki, 1998a) use a method of amortization known as the banker's method. Banker's Queues combine lazy evaluation and memoization to obtain $O(1)$ amortized running times for the *head*, *tail* and *enqueue* operations. The implementation uses lazy lists to achieve lazy evaluation. The basic structure of Banker's Queue is similar to the Batched Queue implementation from Figure 2.1. The difference between the two definitions is that Banker's Queue uses streams as described in Section 3.4.4 instead of lists and maintains the invariant $lenf \leq lenr$.

```
#lang typed/racket
```

Data Structure 2

```
(struct: ( $\alpha$ ) Queue
  ([front : (Stream  $\alpha$ )]
   [lenf : Natural]
   [rear : (Stream  $\alpha$ )]
   [lenr : Natural]))
```

Physicist's Queue

Physicist's Queue (Okasaki, 1998a) uses a method of amortization called the physicist's method. The Physicist's Queue also uses lazy evaluation and memoization to achieve improved amortized running times for its operations. The only drawback of the physicist's method is that it is much more complicated than the banker's method.

Unlike Banker's Queues, Physicist's Queues use suspended lists of type (**Promise** (**Listof** α)) for front instead of streams. Streams are similar to *cons*-lists except that every cell in a stream is delayed. In contrast, suspended lists wrap ordinary *cons*-lists in a single delay. The rear is an ordinary list and is not suspended.

The lengths of the lists are explicitly tracked and guaranteed that the front list is always at least as long as the rear list. Since the front list is delayed, its first element cannot be accessed without executing the entire suspension. So, a working copy of a prefix of the front list is also maintained. This working copy is represented as an ordinary list for efficient access, and is non-empty whenever the front list is non-empty. The following box shows the underlying structure of Physicist's Queue.

#lang typed/racket

Data Structure 3

```
(struct: ( $\alpha$ ) Queue
  ([pref : (Listof  $\alpha$ )]
   [front : (Promise (Listof  $\alpha$ ))]
   [lenf : Natural]
   [rear : (Listof  $\alpha$ )]
   [lenr : Natural]))
```

The Physicist's Queue provides an amortized running time of $O(1)$ for the operations *head*, *tail* and *enqueue*. The implementation of Physicist's Queue is slightly complicated than Banker's Queue though they have the same amortized running times.

Real-Time Queue

Real-Time Queues eliminate the amortization of the Banker's and Physicist's Queues to produce a queue with excellent worst-case as well as amortized running times. Real-Time Queues employ lazy evaluation and a technique called *scheduling* (Okasaki, 1998a) where lazy components are forced systematically so that no suspension takes more than constant time to execute, assuring good asymptotic worst-case running time for the operations on the data structure.

The structure of Real-Time Queues differs from Banker's Queues in three ways. First, Real-Time Queues maintain an extra field of type (**Stream** α) called a *schedule*, which determines when *front* is forced. Second, Real-Time Queues does not maintain length fields as the information is available in *schedule*. And third, *rear* is a list and not a stream.

```
#lang typed/racket
```

```
Data Structure 4
```

```
(struct: ( $\alpha$ ) Queue
  ([front : (Stream  $\alpha$ )]
   [rear : (Listof  $\alpha$ )]
   [schedule : (Stream  $\alpha$ )]))
```

Unlike Physicist's Queues and Banker's Queues which have a amortized running times, Real-Time Queues have an $O(1)$ worst-case running time for the operations *head*, *tail* and *enqueue*.

Implicit Queue

Implicit Queues implement a queue data structure via *implicit recursive slow-down* (Okasaki, 1998a). Implicit recursive slowdown combines laziness with a technique called *recursive slowdown* introduced by Kaplan and Tarjan (1995). This technique is simpler than pure recursive slow-down, but with the disadvantage of amortized rather than worst-case bounds on the running time. Similar to Physicist's Queues and Banker's Queues and unlike Real-Time Queues, Implicit Queues provide an amortized running time of $O(1)$ for the operations *head*, *tail* and *enqueue*.

Bootstrapped Queue

The technique of *bootstrapping* is applicable to problems whose solutions require solutions to simpler instances of the same problem. Bootstrapped Queues use *structural decomposition* (Okasaki, 1998a). In structural decomposition, an implementation that can handle data up to a certain bounded size is used to

implement a data structure that can handle data of unbounded size. Bootstrapped Queues give a worst-case running time of $O(1)$ for the operation *head* and $O(\log^* n)$ for *tail* and *enqueue*. Any queue implementation can be used for bootstrapping. For performance reasons, my implementation of Bootstrapped Queue uses Physicist's Queue for bootstrapping.

Hood-Melville Queue

Hood-Melville Queues (Hood and Melville, 1980) resemble Real-Time Queues in many ways. Both Real-Time and Hood-Melville Queues maintain two lists front and rear and incrementally rotate the elements from rear list to front list when the rear list becomes longer than the front list. The way the elements are rotated is different. Hood-Melville Queues use a more complex technique, called *global rebuilding*. In global rebuilding, reversing of the rear list is done incrementally, a few steps of reversing per normal operation on the data structure. Hood-Melville Queues have worst-case running times of $O(1)$ for the operations *head*, *tail* and *enqueue*.

3.4.2 Deque Implementations

Banker's Deque

Banker's Deques Okasaki (1998a) are double-ended queues with amortized running times. They use the banker's method. The structure of Banker's Deque is similar to the structure of Banker's Queue, using streams and employing the same techniques used in Banker's Queues to achieve amortized running times of $O(1)$ for the operations *head*, *tail*, *last*, *init*, *enqueue-front* and *enqueue*.

Implicit Deque

The techniques used by Implicit Deques are same as that used in Implicit Queues i.e., implicit recursive slowdown (Okasaki, 1998a). Implicit Deques provide

$O(1)$ amortized running times for the operations *head*, *tail*, *last*, *init*, *enqueue-front* and *enqueue*.

Real-Time Deque

Real-Time Deques (Okasaki, 1998a) eliminate the amortization in the Banker's Deque to produce deques with good worst-case behavior. The Real-Time Deques employ the same techniques employed by the Real-Time Queues to provide worst-case running time of $O(1)$ for the operations *head*, *tail*, *last*, *init*, *enqueue-front* and *enqueue*.

3.4.3 Heap Implementations

Binomial Heap

A Binomial Heap (Vuillemin, 1978; Brown, 1978) is a heap-ordered binomial tree. Binomial trees maintains a natural number called its *rank*. The tree structure is defined as follows:

- A Binomial Tree with rank 0 is a leaf with one element.
- A Binomial Tree with rank $r + 1$ is composed of two trees of rank r .

Binomial Heaps support a fast *merge* operation using the special tree structure. Data Structure 5 the underlying structure of Binomial Heaps in Typed Racket.

```
(struct: ( $\alpha$ ) Node
  ([rank : Natural]
   [elem :  $\alpha$ ]
   [trees : (Listof (Node  $\alpha$ ))]))

(struct: ( $\alpha$ ) Heap
  ([compare : ( $\alpha$   $\alpha$   $\rightarrow$  Boolean)]
   [trees : (Listof (Node  $\alpha$ ))]))
```

Data Structure 5

Binomial Heaps provide a worst-case running time of $O(\log n)$ for the operations *insert*, *find-min/max* and *delete-min/max*.

Leftist Heap

Leftist Heaps (Crane, 1972) are heap-ordered binary trees that satisfy the *leftist property*. Each node in the tree is assigned a value called a *rank*. The rank represents the length of its rightmost path from the node in question to the nearest leaf. The leftist property requires that the right descendant of each node has a lower rank than the node itself. As a consequence of the leftist property, the right spine of any node is always the shortest path to a leaf node. Data Structure 6 shows the structure of Leftist Heaps in Typed Racket.

```
(struct: ( $\alpha$ ) Node
  ([rank : Natural]
   [elem :  $\alpha$ ]
   [left : (Tree  $\alpha$ )]
   [right : (Tree  $\alpha$ )]))

(define-type (Tree  $\alpha$ ) (U Null (Node  $\alpha$ )))

(struct: ( $\alpha$ ) Heap
  ([compare : ( $\alpha$   $\alpha$   $\rightarrow$  Boolean)]
   [tree : (Tree  $\alpha$ )])
```

Data Structure 6

The Leftist Heaps provide a worst-case running time of $O(\log n)$ for the operations *insert*, *delete-min/max*, *merge* and a worst-case running time of $O(1)$ for *find-min/max*.

Pairing Heap

Pairing Heaps (Fredman et al., 1986) are a type of heap that simultaneously have a simple implementation and extremely good amortized performance in practice. Unfortunately pairing heaps do not cope well with persistence and

hence can be used only in applications that do not take advantage of persistence. Pairing Heaps are heap-ordered multiway trees represented either as a empty heap or a pair of an element and a list of pairing heaps. Data Structure 7 shows the structure of Pairing Heaps in Typed Racket.

Data Structure 7

```

(struct: ( $\alpha$ ) Node
  ([elem :  $\alpha$ ]
   [trees : (Listof (Tree  $\alpha$ ))]))

(define-type (Tree  $\alpha$ ) (U Null (Node  $\alpha$ )))

(struct: ( $\alpha$ ) Heap
  ([compare : ( $\alpha$   $\alpha$   $\rightarrow$  Boolean)]
   [heap    : (Tree  $\alpha$ )]))

```

Pairing Heaps provide a worst-case running time of $O(1)$ for the operations *insert*, *find-min/max* and *merge*, and an amortized running time of $O(\log n)$ for *delete-min/max*.

Splay Heap

Splay Heaps (Sleator and Tarjan, 1985) are similar to balanced binary search trees. Data Structure 8 shows the structure of Splay Heaps in Typed Racket.

Data Structure 8

```

(struct: ( $\alpha$ ) Node
  ([element :  $\alpha$ ]
   [left    : (Tree  $\alpha$ )]
   [right   : (Tree  $\alpha$ )]))

(define-type (Tree  $\alpha$ ) (U Null (Node  $\alpha$ )))

(struct: ( $\alpha$ ) Heap
  ([compare : ( $\alpha$   $\alpha$   $\rightarrow$  Boolean)]
   [heap    : (Tree  $\alpha$ )]))

```

The difference between a Splay Heap and a balanced binary search tree is that Splay Heaps do not maintain explicit balance information. Instead, every operation on a splay heap restructures the tree with transformations that increase

the balance. Because of the restructuring on every operation, the worst-case running time of all operations is $O(n)$. However, the amortized running time of the operations *insert*, *find-min/max*, *delete-min/max* and *merge* is $O(\log n)$.

Skew Binomial Heap

Skew Binomial Heaps are similar to Binomial Heaps but with a hybrid numerical representation for heaps that is based on the *skew binary numbers* (Myers, 1983). The skew binary number representation is used since incrementing skew binary numbers is quick and simple. Since the skew binary numbers have a complicated addition, the *merge* operation is based on the ordinary binary numbers itself. Skew Binomial Heaps provide a worst-case running time of $O(\log n)$ for the operations *find-min/max*, *delete-min/max* and *merge*, and a worst-case running time of $O(1)$ for the *insert* operation.

Lazy Pairing Heap

Lazy Pairing Heaps (Okasaki, 1998a) are similar to pairing heaps, except that Lazy Pairing Heaps use lazy evaluation. Lazy evaluation is used in this data structure so that the Pairing Heap can cope with persistence efficiently. Analysis of Lazy Pairing Heaps to obtain exact asymptotic running times is difficult, as it is for Pairing Heaps. Lazy Pairing Heaps provide a worst-case running time of $O(1)$ for the operations *insert*, *find-min/max*, and *merge*, and an amortized running time of $O(\log n)$ for the *delete-min/max* operation.

Bootstrapped Heap

Bootstrapped Heaps (Okasaki, 1998a) use a technique of bootstrapping called *structural abstraction*, where one data structure abstracts over a less efficient data structure to get better running times. Bootstrapped Heaps provide a worst-case running time of $O(1)$ for the *insert*, *find-min/max* and *merge* operations and a worst-case running time of $O(\log n)$ for *delete-min/max* operation. Any

heap implementation can be used for bootstrapping. For practical reasons, my implementation of Bootstrapped Heap abstracts over Skew Binomial Heaps.

3.4.4 List Implementations

Binary Random Access List

Binary Random Access Lists abstract the similarities between representations of the numbers and lists to derive a representation of list structure. The Binary Random Access List representation abstracts over the binary numerical representation (Okasaki, 1998a) to achieve a worst-case running time of $O(\log n)$ for its random-access operations *list-ref* and *list-set*. Data Structure 9 shows the structure of the structure of Binary Random Access Lists in Typed Racket.

```
(struct: ( $\alpha$ ) Leaf ([first :  $\alpha$ ]))
```

Data Structure 9

```
(struct: ( $\alpha$ ) Node
  ([first :  $\alpha$ ]
   [left  : (Tree  $\alpha$ )]
   [right : (Tree  $\alpha$ )]))
```

```
(define-type (Tree  $\alpha$ ) (U (Leaf  $\alpha$ ) (Node  $\alpha$ )))
```

```
(struct: ( $\alpha$ ) Root
  ([size : Integer]
   [first : (Tree  $\alpha$ )]
   [rest  : (List  $\alpha$ )]))
```

```
(define-type (List  $\alpha$ ) (U Null (Root  $\alpha$ )))
```

Binary Random Access Lists provide a worst-case running time of $O(\log n)$ for the operations *cons*, *first* and *rest* contrary to Typed Racket's built-in list's worst-case running time of $O(1)$ for the same operations.

Skew Binary Random Access List

Skew Binary Random Access Lists are similar to Binary Random Access Lists, but use the skew binary number representation, improving the running times of some operations. Unlike Binary Random Access Lists, that have worst-case running time of $O(\log n)$, Skew Binary Random Access Lists provide worst-case running time of $O(1)$ for the operations *cons*, *head* and *tail*. And like Binary Random Access Lists, Skew Binary Random Access Lists also provide worst-case running time of $O(\log n)$ for *list-ref* and *list-set* operations.

VList

VLists (Bagwell, 2002) resemble normal cons lists but provide efficient versions of many operations that are much slower on standard lists. They combine the extensibility of linked lists with the fast random access capability of arrays. Data Structure 10 shows the structure of VLists in Typed Racket. The *elements* field in the *Base* in Data Structure 10 is a random access list.

<pre>#lang typed/racket (struct: (α) Base ([previous : (Block α)] [elements : (List α)])) (define-type (Block α) (U Null (Base α))) (struct: (α) VList ([offset : Integer] [base : (Base α)] [size : Integer]))</pre>	<div style="border: 1px solid black; padding: 2px 5px; display: inline-block;">Data Structure 10</div>
---	--

VLists provide worst-case running times of $O(1)$ for the operations *cons*, *head* and *tail*, *list-ref* and *list-set* operations. This VList implementation is built internally on Skew Binary Random Access Lists. VLists provide the standard Random Access List API operations.

Catenable List

Catenable Lists (Okasaki, 1995) are a list data structure with an efficient append operation, achieved using the bootstrapping technique of *structural abstraction*. Catenable Lists are abstracted over Bootstrapped Queue. They have an amortized running time of $O(1)$ for the basic list operations and for the operations *cons-to-end* and *append*.

Streams

Streams (Okasaki, 1998a) are also known as lazy lists. They are similar to ordinary lists and provide a similar API. Stream API provides *stream*, *stream-car*, *stream-cdr* and *stream-cons* which are similar to *list*, *first*, *rest* and *cons* from the list interface respectively. Along with these basic functions, Stream API also provide some utility functions. Many data structures implemented in this library use Streams to achieve lazy evaluation. Streams do not change the asymptotic performance of any list operations, but introduce overhead at each suspension. And forcing a suspension takes no more than $O(1)$ and hence *stream-car*, *stream-cdr* and *stream-cons* have a running time of $O(1)$. Since streams have distinct evaluation behavior, they are given a distinct type, (**Stream** α).

3.4.5 Other Data Structures

Hash Lists

Hash Lists (Bagwell, 2002) are similar to association lists, and are implemented using a modified VList structure. The modified VList contains two components: the data and the hash table. Both components grow as the hash list grows. The running time for Hash Lists operations such as *insert*, *delete*, and *lookup* are close to those for standard chained hash tables.

Tries

A Trie (also known as a Digital Search Tree) (Okasaki, 1998a) is a data structure that takes advantage of the structure of aggregate types to achieve good running times for its operations. Our implementation provides Tries in which the keys are lists of the element type; this is sufficient for representing many aggregate data structures. In our implementation, each trie is a multiway tree with each node of the multiway tree carrying data of base element type. Tries provide *lookup* and *insert* operations with better asymptotic running times than hash tables.

Red-Black Tree

Red-Black Trees (Bayer, 1972) are a classic data structure, consisting of a binary search trees in which every node is colored either red or black, according to the following two balance invariants:

- no red node has a red child, and
- every path from root to an empty node has the same number of black nodes.

The two invariants together guarantee that the longest possible path with alternating black and red nodes, is no more than twice as long as the shortest possible path, with black nodes only. This property helps achieve good running times for the tree operations. This implementation is based on Okasaki (1999). Data Structure 11 shows the structure of Red-Black Trees in Typed Racket.

```
#lang typed/racket

(define-type Color (U 'red 'black))

(struct: ( $\alpha$ ) RedBlackNode
  ([color : Color]
   [element :  $\alpha$ ]
   [left : (Tree  $\alpha$ )]
   [right : (Tree  $\alpha$ )]))

(define-type (Tree  $\alpha$ ) (U Null (RedBlackNode  $\alpha$ )))

(struct: ( $\alpha$ ) RedBlackTree
  ([compare : ( $\alpha$   $\alpha \rightarrow$  Boolean)]
   [tree : (Tree  $\alpha$ )]))
```

Data Structure 11

The operations *member?*, *insert* and *delete*, which respectively check membership, insert and delete elements from the tree, have worst-case running time of $O(\log n)$. Red-Black Trees have the type (**RedBlackTree** α).

Treap

Treaps (Seidel and Aragon, 1996) are binary search trees in which each node has both a search key and a priority. Its keys are sorted in-order and the priority of each node is lower than the priorities of its children. Because of these properties, a treap is a binary search tree for the keys and a heap for its priorities. Assigning random priorities to the nodes of the treap results in better balance in the treap. Hence treaps are also known as randomized binary search trees. Data Structure 12 shows the underlying structure of Treaps in Typed Racket.

```
#lang typed/racket
```

Data Structure 12

```
(struct: ( $\alpha$ ) Node
  ([element :  $\alpha$ ]
   [left    : (Tree  $\alpha$ )]
   [right   : (Tree  $\alpha$ )]
   [priority : Real]))

(define-type (Tree  $\alpha$ ) (U Null (Node  $\alpha$ )))

(struct: ( $\alpha$ ) Treap
  ([compare : ( $\alpha$   $\alpha$   $\rightarrow$  Boolean)]
   [tree    : (Tree  $\alpha$ )]
   [size    : Integer]))
```

Treaps implement a worst case running time of $O(\log n)$ for the operations *insert*, *find-min/max* and *delete-min/max*. A Treap has the type **(Treap α)**.

Performance Analysis

This chapter reports on my performance evaluation of the library using micro-benchmarks. The results demonstrate the practical usefulness of purely functional data structures. The benchmarks compare the performance of my purely functional data structures with each other and with simple functional implementations based on lists and with imperative implementations already available as Racket libraries. All the data structures are implemented in Typed Racket, with the exception of the pre-existing imperative versions which are implemented in untyped Racket. Data structures in untyped Racket and Typed Racket are benchmarked in their respective languages to ensure comparability; otherwise language boundary-crossing would impose substantial contract overheads.

The benchmarking was done on a 2.1 GHz Intel Core 2 Duo (Linux) machine using Racket version 5.0.2. Each benchmark was run 10 times, with times measured by the Racket's *time* form. In the tables below, all times are milliseconds of CPU time as reported by Racket, including garbage collection time.

4.1 Queue Performance

Table 4.1 shows the performance of the Physicist's Queue, Banker's Queue, Real-Time Queue and Bootstrapped Queue compared with a naive implementation based on lists and an imperative queue from the Racket standard library. Some functions of the imperative queue from the Racket standard library provide contracts. In the benchmarks where these functions are used, separate

¹The constructor functions *queue*, *heap* and *list* were repeated only 100 times.

benchmarks for the implementation without contracts has been provided. The *Size* column in Table 4.1 indicates the initial size of the queue and the times are the time taken for performing each operation 100000 times on queues of different initial sizes, averaged over 10 runs. Because the *head* operation runs in a very short amount of time, it is repeated 1000000 times to reduce noise in the benchmark.

Benchmark Code 1 shows the Typed Racket code used for generating the benchmarks for the *enqueue* operation; the untyped Racket benchmarks are run with an equivalent untyped script.

<pre>(define <i>Size</i> 1000) (: <i>que</i> : (Queue Integer)) (define <i>que</i> (<i>build-queue</i> <i>Size</i> <i>add1</i>)) (: <i>list</i> : (Listof Integer)) (define <i>list</i> (<i>build-list</i> 100000 <i>add1</i>)) (<i>time</i> (<i>foldl enqueue que list</i>))</pre>	<div style="border: 1px solid black; padding: 2px 5px; display: inline-block;">Benchmark Code 1</div>
---	---

Table 4.1 suggests four observations:

1. For the queue constructor *queue*, Physicist's Queues and Bootstrapped Queues perform slightly better than the imperative queues, the performance of imperative queues is better than Banker's Queues, and the queue implementation based on lists outperforms all the other implementations.
2. For the *head* operation, all the implementations perform better than Real-Time Queues. Performance of Physicist's Queues, Bootstrapped Queues and imperative queues are similar and slightly better than Banker's Queues. The queue implementation based on lists performs slightly better than all of these implementations.
3. For the *enqueue* operation, the implementation based on lists is extremely slow when compared to other implementations as expected. Physicist's

Size	Operation	Banker's	Physicist's	Real-Time	Bootstrapped	List	Imperative
1000	<i>queue</i>	72	16	137	20	6	24
	<i>head</i>	55	45	85	40	30	55
	<i>enqueue</i>	127	10	176	22	256450	12
10000	<i>queue</i>	887	232	1576	227	61	290
	<i>head</i>	55	45	95	45	35	60
	<i>enqueue</i>	132	11	172	18	314710	14
100000	<i>queue</i>	13192	3410	20332	2276	860	3590
	<i>head</i>	60	40	90	50	35	60
	<i>tail</i> [†]	312	412	147	20	7	10
	<i>enqueue</i>	72	12	224	18	1289370	14
1000000	<i>queue</i>	182858	65590	294310	53032	31480	68310
	<i>head</i>	60	40	90	40	35	60
	<i>tail</i> [†]	1534	243	1078	20	8	10
	<i>enqueue</i>	897	30	1218	20	∞ [‡]	16

[†] Since 100000 (successive) *tail* (or *dequeue*) operations cannot be performed on 1000 and 10000 element queue, the *tail* operation is omitted for these sizes.

[‡] Longer than 30 minutes.

Figure 4.1: Queue Performance: Individual Operations

Queues, Bootstrapped Queues and imperative queues have similar performance and perform better than Banker's Queue and Real-Time Queues.

4. For the *tail* operation, performance of imperative queues and the implementation based on lists are comparable and is slightly better than Bootstrapped Queues. All the other functional data structures are slower than these queue implementations.

Table 4.2 presents the results of a multiple-operation workload micro benchmarks. The times in are for building a queue by repeating the *enqueue* operation *N* times and then performing *head* and *tail* on the resulting queue *N* times with *N* taking the values 1000000, 2000000, 3000000, 4000000 and 5000000, again averaged over 10 runs. Benchmark Code 2 shows the Typed Racket code used to benchmark functional queues in Typed Racket. Untyped benchmark code in Racket similar to Benchmark Code 2 was used to benchmark the imperative queue implementation.

# of Repetitions	Banker's	Physicist's	Real-Time	Bootstrapped	Imperative	Imperative [‡]
1000000	2040	2010	2170	560	750	410
2000000	5060	5040	4430	1120	1490	810
3000000	7990	7880	7260	2180	2260	1190
4000000	10240	10280	9230	2930	3780	1630
5000000	13690	13670	11930	3670	4580	2140

[‡] Imperative Queue implementation without contracts.

Figure 4.2: Queue Performance: Multiple Operations

The results of the benchmarks from the Table 4.2 indicate that the performance of Bootstrapped Queues is faster than the imperative queue implementation with contracts and the imperative queue implementation without contracts perform better than Bootstrapped Queues. Real-Time Queues, Banker's Queues and Physicist's Queues are all almost 3 times slower than imperative queues.

```
(: benchmark : Integer → Integer)
(define (benchmark N)

  (: build : Integer (Queue Integer) → (Queue Integer))
  (define (build i que)
    (if (<= i N)
        (build (add1 i) (enqueue i que))
        que))

  (: q : (Queue Integer))
  (define q (build 0 (queue)))

  (let add-all ([sum 0] [que q])
    (if (empty? que)
        sum
        (add-all (+ sum (head que)) (tail que)))))

(time (benchmark 1000000))
```

Benchmark Code 2

According to Okasaki (1998a) Real-Time Queues are among the fastest queue implementations. However, the above results show that the performance of Typed Racket implementation of Real-Time Queues is slower than the other

functional queue implementations. The performance of some operations on Physicist's Queue is comparable to the performance of Bootstrapped and imperative Queues. But the overall performance of Bootstrapped Queue is better than Physicist's Queue, the imperative queue implementation with contracts and the other functional queue implementations. The imperative queue implementation without contracts performs better than all the functional queue implementations.

4.2 Heap Performance

Table 4.3 shows the performance of the Leftist Heap, Pairing Heap, Binomial Heap and Bootstrapped Heap implementations, compared with a simple implementation based on sorted lists, and a simple imperative heap. The *Size* column of the table indicates the initial size of the heaps. The times in the table are time taken for performing heap operations 100000 times on heaps with different initial sizes, averaged over 10 runs. Again, because the *find* operation runs in a very short amount of time, it is repeated 1000000 times to reduce noise in the benchmark.

The performance characteristics of the heap implementations can be derived from the results of the benchmarks from the Table 4.3.

1. For the heap constructor *heap*, Pairing Heaps and the imperative heap implementation perform similarly for smaller sizes and for larger sizes imperative heaps perform better than all the functional heap implementations. Pairing Heaps perform better than all the other functional heap implementations and the heap implementation based on lists outperforms all other implementations.
2. For the *insert* operation, the heap implementation based on lists is extremely slow. Among other heap implementations, the performance of imperative heaps, Binomial Heaps and Pairing Heaps is almost the same and is better than the other functional heap implementations.

Size	Operation	Binomial	Leftist	Pairing	Bootstrapped	List	Imperative
1000	<i>heap</i>	45	192	30	122	9	30
	<i>insert</i>	36	372	24	218	323874	20
	<i>find</i>	480	40	40	45	35	40
10000	<i>heap</i>	422	2730	260	1283	76	360
	<i>insert</i>	34	358	28	224	409051	24
	<i>find</i>	430	40	45	45	35	40
100000	<i>heap</i>	6310	40580	4240	24418	1010	3490
	<i>insert</i>	33	434	30	198	1087545	30
	<i>find</i>	480	40	45	40	40	45
	<i>delete</i> [†]	986	528	462	1946	7	180
1000000	<i>heap</i>	109380	471588	80210	293788	11140	43010
	<i>insert</i>	32	438	28	218	∞ [‡]	140
	<i>find</i>	590	45	40	45	40	45
	<i>delete</i> [†]	1488	976	1489	3063	8	280

[†] Since 100000 (successive) *delete* operations cannot be performed on 1000 and 10000 element heap, running time for *delete* operation for these sizes are not available.

[‡] Takes longer than 30 minutes.

Figure 4.3: Heap Performance: Individual Operations

3. For the *find* operation, the performance of almost all the heap implementations is very similar, except for Binomial Heaps, which are slower than the other heap implementations.
4. For the *delete* operation, the heap implementation based on lists outperforms the other heap implementations. Imperative heaps perform better than all the functional heap implementations. Among functional heaps, Leftist Heaps perform slightly better than Binomial and Pairing Heaps.

The times in Table 4.4 are the times taken for building a heap by repeating the *insert* operation *N* times and then performing *find-min/max* and *delete-min/max* on the resulting queue *N* times, with *N* taking the values 100000, 200000, 300000, 400000 and 500000. Benchmark Code 3 shows the Typed Racket code used to benchmark functional heaps in Typed Racket. Untyped benchmark code in Racket similar to Benchmark Code 3 was used to benchmark the imperative heap implementation in Racket.

The results of the benchmarks from the Table 4.4 show that Pairing Heaps perform slightly better than imperative heaps, and the imperative heaps better than all other heap implementations.

Benchmark Code 3

```

(benchmark : Integer → Integer)
(define (benchmark N)

  (build : Integer (Heap Integer) → (Heap Integer))
  (define (build i heap)
    (if (<= i N)
      (build (add1 i) (insert i heap))
      heap))

  (init-heap : (Heap Integer))
  (define init-heap (build 0 (heap < 1)))

  (let add-all ([sum 0] [init-heap init-heap])
    (if (empty? init-heap)
      sum
      (add-all (+ sum (find-min/max init-heap))
        (delete-min/max init-heap))))))

(time (benchmark 100000))

```

# of Repetitions	Binomial	Leftist	Pairing	Bootstrapped	Imperative
100000	290	620	120	320	240
200000	610	1330	320	660	480
300000	980	2140	480	1030	840
400000	1310	3020	690	1370	940
500000	1630	3760	820	1660	1410

Figure 4.4: Heap Performance: Multiple Operations

The results show that some functional heap implementations perform comparable to and in some cases better than imperative heap implementation. Among the functional heap implementations, Pairing Heaps are fastest overall.

4.3 List Performance

Table 4.5 shows the performance of Skew Binary Random Access Lists and VLists compared with built-in lists. The *Size* column in the table indicate the initial size of the list. The times in the table indicate the time taken to perform list operations 100000 times on lists of different initial sizes. The results of the benchmark show the performance characteristics of each list implementation.

Size	Operation	RAList	VList	List
1000	<i>list</i>	24	51	2
	<i>list-ref</i>	77	86	240
	<i>first</i>	2	9	1
	<i>rest</i>	20	48	1
	<i>last</i>	178	40	520
10000	<i>list</i>	263	476	40
	<i>list-ref</i>	98	110	2538
	<i>first</i>	2	9	1
	<i>rest</i>	9	28	1
	<i>last</i>	200	52	5414
100000	<i>list</i>	2890	9796	513
	<i>list-ref</i>	124	131	33187
	<i>first</i>	3	10	1
	<i>rest</i>	18	40	1
	<i>last</i>	204	58	77217
1000000	<i>list</i>	104410	147510	4860
	<i>list-ref</i>	172	178	380960
	<i>first</i>	2	10	1
	<i>rest</i>	20	42	1
	<i>last</i>	209	67	755520

Figure 4.5: List Performance

1. For the list constructor *list*, *cons* list perform better than RALists and VLists, RALists perform much better than VLists.
2. For *first*, the performance of RALists and *cons* lists is similar and slightly better than VLists.

3. For *rest*, the performance of *cons* list is better than RALists and the performance of RALists is better than VLists.
4. For *list-ref*, the performance of cons lists is slow compared to RALists and VLists, and the performance of RALists and VLists is similar.
5. For *last*, the performance of cons lists is slow compared to RALists and VLists. And the performance of VLists is better than RALists.

The analysis of the list benchmarks show that the built-in lists are faster than VList and RAList implementations for the operations *first*, *rest* and *cons*. Unsurprisingly, in the case of the random-access operations, RALists and VLists perform better than the built-in list implementation.

4.4 Conclusion

The analysis shows that Bootstrapped Queues and Pairing Heaps are among the fastest Typed Racket functional queue and heap implementations respectively. Choosing these functional queue and heap data structures produces results close to, and in some cases better than optimized imperative versions of these data structures. Therefore, these functional queues and heaps serve as viable alternatives to imperative implementations. For some basic operations, naive implementations based on lists perform significantly better than more sophisticated functional and imperative data structures. However, other operations perform extremely poorly when using lists, making them impractical for general use. In summary, for queues and heaps programmers can freely use functional data structures without worrying that they lose performance.

Despite the disadvantage of library implementation, the performance of the basic operations of RALists and VLists competitive with built-in *cons*-lists. For the simple operations, the data structures of this thesis perform acceptably for general use, and for a few operations, the implementations of the library are significantly faster. This makes RALists and VLists viable options when fast and growable random access data structures are required.

Evaluation of Typed Racket

This chapter reports on my experience with Typed Racket’s type system in the context of purely functional data structures and contrasts it with other statically typed functional languages. The data structures developed to support this thesis were originally developed in statically typed functional programming languages with different type systems. Thus, eventually the suitability of Typed Racket for implementing purely functional data structures serves as a valuable test case for evaluating Typed Racket’s type system against other type systems.

5.1 Benefits of Typed Racket

Data Structures from Other Languages

The purpose of Typed Racket is to facilitate *the gradual porting of untyped code to typed sister languages*. Typed Racket was developed as a sister language of Racket and support the Racket idioms. However, all the data structures developed to support this thesis, were originally developed in other statically typed functional languages such as ML and Haskell. These languages have different type system and support different idioms than Racket.

The following examples highlight the similarities between Typed Racket and ML code by comparing the data definitions and function definitions in these two languages. Although, the Typed Racket’s type system is different from other statically typed languages, the definitions in these statically typed languages have a straightforward mapping to Typed Racket.

Figure 5.1 and 5.2 show the Banker’s Queue and Physicist’s Queue structure definitions in Typed Racket and ML respectively.

<pre>(struct: (α) Queue ([lenf : Integer] [front : (Stream α)] [lenr : Integer] [rear : (Stream α)])</pre>	<pre>type ' α Queue = int * ' α Stream * int * ' α Stream</pre>
---	---

Figure 5.1: Typed Racket and ML Definition: Banker's Queue

<pre>(struct: (α) Queue ([pref : (Listof α)] [lenf : Integer] [front : (Promise (Listof α))] [lenr : Integer] [rear : (Listof α)])</pre>	<pre>type ' α Queue = ' α list * int * ' α list susp * int * ' α list</pre>
--	---

Figure 5.2: Typed Racket and ML Definition: Physicist's Queue

The *tagged unions* provided by languages like ML and Haskell can be emulated using the union operator **U** in Typed Racket. Consider the definitions of a binary tree in Figure 5.3.

<pre>(define-type (Tree α) (U Leaf (Node α))) (struct: Leaf ()) (struct: (α) Node ([data : α] [left : (Tree α)] [right : (Tree α)])</pre>	<pre>datatype ' α tree = Node of {data : ' α, left : ' α tree, right : ' α tree} Leaf;</pre>
---	--

Figure 5.3: Typed Racket and ML Definition: Binary Tree

Although tagged unions are not a part of Typed Racket, emulating ML definitions using tagged unions is simple and straightforward.

Pattern matching is a widely used technique in statically typed functional languages like ML and Haskell. Figure 5.4 shows the ML function definition to balance a Red-Black Tree (Okasaki, 1999).

```

fun balance T (B, T(R, T(R, a, x, b), y, c), z, d) = T(R, T(B, a, x, b), y, T(B, c, z, d))
| balance T (B, T(R, a, x, T(R, b, y, c)), z, d) = T(R, T(B, a, x, b), y, T(B, c, z, d))
| balance T (B, a, x, T(R, T(R, b, y, c), z, d)) = T(R, T(B, a, x, b), y, T(B, c, z, d))
| balance T (B, a, x, T(R, b, y, T(R, c, z, d))) = T(R, T(B, a, x, b), y, T(B, c, z, d))
| balance T body = T body

```

Figure 5.4: Pattern Matching in ML

balance uses the pattern matching technique. The function takes a tree as input and returns a tree. The function definition has five clauses. The structure of the input tree is matched against the left side of the clauses and upon a match, the right side is returned.

Typed Racket provides a sophisticated match construct for pattern matching known as **match**. It supports a wide variety of useful pattern-matching forms and makes porting code from ML to Typed Racket straightforward. Figure 5.5 shows the Typed Racket definition of *balance* using **match**. It is similar to the ML definition of *balance*.

```

(: balance : (∀ (α) ((Tree α) → (Tree α))))
(define (balance tree)
  (match tree
    [(T B (T R (T R a x b) y c) z d) (T R (T B a x b) y (T B c z d))]
    [(T B (T R a x (T R b y c)) z d) (T R (T B a x b) y (T B c z d))]
    [(T B a x (T R (T R b y c) z d)) (T R (T B a x b) y (T B c z d))]
    [(T B a x (T R b y (T R c z d))) (T R (T B a x b) y (T B c z d))]
    [else tree]))

```

Figure 5.5: Pattern Matching in Typed Racket

```
#lang typed/racket
(require typed/test-engine/racket-tests)
(require "bankers-queue.rkt")
(check-expect (head (queue 4 5 2 3)) 4)
(check-expect (tail (queue 4 5 2 3))
              (queue 5 2 3))
```

Figure 5.6: Examples of Unit Tests

Other Features of Typed Racket

The type error messages in Typed Racket are clear and easy to understand. The type checker highlights precise locations that are responsible for type errors. This makes it very easy to debug the type errors.

Typed Racket comes with a unit testing framework, which makes it simple to write tests. Figure 5.6 shows the use of Typed Racket’s testing framework. The *check-expect* form takes the actual and expected value, and compares them, printing a message at the end summarizing the results of all tests.

The introductory and reference manuals of Racket in general and Typed Racket in particular are comprehensive and quite easy to follow and understand.

5.2 Disadvantages of Typed Racket

Even though my overall experience with Typed Racket is positive, there are several negative aspects to programming with Typed Racket.

Polymorphic Recursion

Most significantly for this work, Typed Racket does not support polymorphic non-uniform recursive datatype definitions. Number of data structures by Okasaki

(1998a) extensively use polymorphic recursion. Because of this limitation, many definitions had to be first converted to uniform recursive datatypes before being implemented. For instance, consider the definition of *Seq* structure in Structure 1. This definition is not allowed by Typed Racket.

```
#lang typed/racket
```

Structure 1

```
(struct: ( $\alpha$ ) Seq
  ([elem :  $\alpha$ ]
   [recur : (Seq (Pair A A))]))
```

The definition must be converted not to use polymorphic recursion. Structure 2 shows the converted *Seq* structure.

```
#lang typed/racket
```

Structure 2

```
(struct: ( $\alpha$ ) Elem ([elem :  $\alpha$ ]))

(struct: ( $\alpha$ ) Pare
  ([pair : (Pair (EP  $\alpha$ ) (EP  $\alpha$ ))]))

(define-type (EP  $\alpha$ ) (U (Elem  $\alpha$ ) (Pare  $\alpha$ )))

(struct: ( $\alpha$ ) Seq
  ([elem : (EP  $\alpha$ )]
   [recur : (Seq  $\alpha$ )]))
```

Unfortunately, this translation introduces the possibility of illegal states that the type checker is unable to rule out.

Variable-arity Functions

Consider the two *foldr* expressions in Figure 5.7. The list in first expression is the the built-in *cons*-list and the the list in the second expression is a *VList* from my purely functional data structure library. Although the two expressions are identical, the results produced by the two expressions are different. The results are 2 and -14 respectively.

```
#lang typed/racket

(foldr - 1 (list 1 2 3 4 5))

(require (prefix-in v: (planet krhari/pfds:1:5/vlist)))
(v:foldr - 1 (v:list 1 2 3 4 5))
```

Figure 5.7: Variable-arity Functions

The difference between the two results is because it is currently not possible to correctly type Racket functions such as *foldr* and *foldl* because of the limitations of Typed Racket’s handling of variable-arity functions (Strickland et al., 2009).

User-Defined Data Types

Although Racket supports extension of the behavior of primitive operations such as printing and equality on user-defined data types, Typed Racket currently does not support this. Thus, it is not possible to compare any of our data structures accurately using *equal?*, and they are printed opaquely.

Typed Racket allows programmers to name arbitrary type expressions with the **define-type** form. However, the type printer does not take into account definitions of polymorphic type aliases when printing types, leading to the internal implementations of some types being exposed. This makes the printing of types confusingly long and difficult to understand, especially in error messages.

Racket’s Numeric Tower

Typed Racket provides precise types for Racket’s numeric tower. Because of the precise numeric types, the type for functions like *+* and *-* have several cases for each type in the numeric hierarchy. The type errors involving these functions contain the whole numeric type hierarchy and leads to illegible error messages.

```

Type Checker: Polymorphic function foldl could not be applied to arguments:
Domains: (a b → b) b (Listof a)
          (a b c → c) c (Listof a) (Listof b)
          (a b c d → d) d (Listof a) (Listof b) (Listof d)
Arguments:
(case-lambda
  (Exact-Positive-Integer Exact-Nonnegative-Integer * → Exact-Positive-Integer)
  (Exact-Nonnegative-Integer Exact-Positive-Integer
   Exact-Nonnegative-Integer * → Exact-Positive-Integer)
  (Exact-Nonnegative-Integer * → Exact-Nonnegative-Integer)
  (Integer * → Integer)
  (Exact-Rational * → Exact-Rational)
  (Nonnegative-Float * → Nonnegative-Float)
  (Float * → Float)
  ((U Nonnegative-Float Exact-Positive-Integer Zero) * → Nonnegative-Float)
  (Float Real * → Float)
  (Real Float Real * → Float)
  (Inexact-Real * → Inexact-Real)
  (Real * → Real)
  ((U Float-Complex Inexact-Real Exact-Rational) * → Float-Complex)
  (Float-Complex Complex * → Float-Complex)
  (Complex Float-Complex Complex * → Float-Complex)
  (Complex * → Complex)) Zero (List Positive-Fixnum String)

```

Figure 5.8: Type Error: *foldl*

Consider

```
(foldl + 0 (list 1 "2"))
```

In the above expression, since the input list contains a string and the function `+` is not defined for strings, Typed Racket, appropriately, signals a type error. But the error message is illegible. Figure 5.8 shows the error message displayed by Typed Racket.

Type Checker: Polymorphic function vector-append could not be applied to arguments:
*Domain: (Vectorof a) **
Arguments: (Vectorof Exact-Positive-Integer) (Vectorof Integer)

Figure 5.9: Type Error: *vector-append*

Local Type Inference

Typed Racket's use of local type inference also leads to spurious type errors, especially in the presence of precise types for Racket's numeric hierarchy. For example, Typed Racket distinguishes integers from positive integers, leading to a type error in the following expression:

(vector-append (vector -1 2) (vector 1 2))

since the first vector and second vector have the type *(Vectorof Exact-Positive-Integer)* and *(Vectorof Integer)* respectively, neither of which is a subtype of the other. Because of this problem, Typed Racket throws the error shown in Figure 5.9.

Working around this requires manual annotation to ensure that both vectors have element type *Integer*.

CHAPTER 6

Related Work

In the past two decades many new efficient functional data structures have been developed. Some of the purely functional data structures implemented for Typed Racket in this thesis are available in other functional languages such as ML, OCaml, Haskell and Clojure. All the data structures implemented for this work were originally implemented in one of these languages. In addition, some of the data structures were available in plain Racket.

6.1 In Other Languages

Original implementations of the data structures presented by Okasaki (1998a) are in ML and Haskell (Okasaki, 1998b). The implementation of these data structures are also available in OCaml (Mottl, 2008). VLists and others (Bagwell, 2002) were implemented in a variant of Lisp and OCaml.

The built-in data structures available in Clojure (Hickey, 2008), a new dialect of Lisp for the Java Virtual Machine, are functional data structures. For example, the hash map/set and vector implementations available in Clojure are based on array mapped hash tries (Bagwell, 2001).

6.2 In Racket

Racket's existing collection of user-provided libraries, PLaneT (Matthews, 2006), contains an implementation of Random Access Lists ¹, an implementation of

¹<http://planet.plt-scheme.org/display.ss?package=ralist.plt&owner=dvanhorn>

pairing heaps ², as well as a collection of a small number of functional data structures ³. The functional data structures developed for this thesis provide more utility functions and also the benchmarks show that they perform better than the functional data structures available from PLaneT.

²<http://planet.plt-scheme.org/display.ss?package=pairing-heap.plt&owner=wmfarr>

³<http://planet.plt-scheme.org/display.ss?package=galore.plt&owner=soegaard>

Conclusion

Programming languages require support for efficient data structures. In particular, the data structures should support programming in the paradigm that the language supports. Racket and its statically typed dialect Typed Racket are functional languages, but lack many functional data structures. In this thesis, I have described a comprehensive library of efficient functional data structures for Typed Racket and an evaluation of Typed Racket as a language for developing large libraries of typed programs.

7.1 Contributions

In support of this MS thesis, I implemented a comprehensive library of data structures in Typed Racket. Chapter 4 shows that the functional data structures are as efficient as their imperative counterparts, supporting efficient programming in the functional paradigm.

The evaluation of Typed Racket shows that even though Typed Racket has a type system built to support the Racket idioms, it supports straightforward porting of code from other statically typed functional languages with different type systems, such as Haskell and ML. Additionally Typed Racket is easy to learn and use.

7.2 Future Work

As mentioned earlier, because Typed Racket does not support polymorphic recursion, it is not possible to implement some data structures, such as finger trees

(Hinze and Paterson, 2006). Support for polymorphic recursion will be useful. It may also improve the performance of some of the existing data structures.

The library of data structures developed is a comprehensive foundation for implementing data structures that support parallel programming. It remains to be seen how these data structures perform in Typed Racket and much more work remains to be done in evaluating Typed Racket in the context of these data structures that support parallel programming.

Bibliography

- C. Aragon and R. Seidel. Randomized Search Trees. *Annual IEEE Symposium on Foundations of Computer Science*, pages 540–545, 1989.
- P. Bagwell. Fast And Space Efficient Trie Searches. Technical Report LAMP-REPORT-2000-001, Ecole Polytechnique Federale de Lausanne, 2000.
- P. Bagwell. Ideal Hash Trees. *Es Grands Champs*, 1195, 2001.
- P. Bagwell. Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays. In *Implementation of Functional Languages, 14th International Workshop*, pages 34–50, 2002.
- R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.
- M. R. Brown. Implementation and Analysis of Binomial Queue Algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.
- C. A. Crane. Linear lists and priority queues as balanced binary trees. PhD thesis, Stanford University, Stanford, CA, USA, 1972.
- M. L. Fredman, R. Sedgwick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
- R. Hickey. The Clojure programming language. In *Proceedings of the Symposium on Dynamic languages*, DLS '08, pages 1:1–1:1, 2008.
- R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16:197–217, 2006.
- R. T. Hood and R. C. Melville. Real Time Queue Operations in Pure LISP. Technical report, Cornell University, Ithaca, NY, USA, 1980.

- H. Kaplan and R. E. Tarjan. Persistent lists with catenation via recursive slow-down. In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 93–102, New York, NY, USA, 1995. ACM.
- J. Matthews. Component Deployment with PLaneT You Want it Where? In *Workshop on Scheme and Functional Programming*, pages 157–165, 2006.
- M. Mottl. Purely Functional Data Structures in OCaml. http://www.ocaml.info/home/ocaml_sources.html#toc20, 2008.
- E. W. Myers. An Applicative Random-Access Stack. *Information Processing Letters*, 17(5):241–248, 1983.
- C. Okasaki. Amortization, Lazy Evaluation, and Persistence: Lists with Catenation via Lazy Linking. In *IEEE Symposium on Foundations of Computer Science*, pages 646–654. IEEE Computer Society Press, 1995.
- C. Okasaki. Purely Functional Data Structures. Cambridge University Press, 1998a. ISBN 0521663504.
- C. Okasaki. Purely Functional Data Structures in ML and Haskell. <http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#cup98>, 1998b.
- C. Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.
- R. Seidel and C. R. Aragon. Randomized Search Trees. In *Algorithmica*, pages 540–545, 1996.
- D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.
- T. S. Strickland, S. Tobin-Hochstadt, and M. Felleisen. Practical Variable-Arity Polymorphism. In *Programming Languages and Systems: ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 32–46, 2009.

- S. Tobin-Hochstadt. Typed Scheme: From Scripts to Programs. PhD thesis, Northeastern University, Boston, MA, USA, 2010.
- S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. pages 395–406, 2008.
- J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.