Functional Data Structures for Typed Racket

Hari Prashanth K R
Northeastern University
krhari@ccs.neu.edu

Sam Tobin-Hochstadt
Northeastern University
samth@ccs.neu.edu

Abstract

Scheme provides excellent language support for programming in a functional style, but little in the way of library support. In this paper, we present a comprehensive library of functional data structures, drawing from several sources. We have implemented the library in Typed Racket, a typed variant of Racket, allowing us to maintain the type invariants of the original definitions.

1. Functional Data Structures for a Functional Language

Functional programming requires more than just `lambda`; library support for programming in a functional style is also required. In particular, efficient and persistent functional data structures are needed in almost every program.

Scheme does provide one supremely valuable functional data structure—the linked list. This is sufficient to support many forms of functional programming (Shivers 1999) (although lists are sadly mutable in most Schemes), but not nearly sufficient. To truly support efficient programming in a functional style, additional data structures are needed.

Fortunately, the last 15 years have seen the development of many efficient and useful functional data structures, in particular by Okasaki (1998) and Bagwell (2002; 2000). These data structures have seen wide use in languages such as Haskell and Clojure, but have rarely been implemented in Scheme.

In this paper, we present a comprehensive library of efficient functional data structures, implemented in Typed Racket (Tobin-Hochstadt and Felleisen 2008), a recently-developed typed dialect of Racket (formerly PLT Scheme). The remainder of the paper is organized as follows. We first present an overview of Typed Racket, and describe how typed functional datastructures can interoperate with untyped, imperative code. Section 2 describes the data structures, with their API and their performance characteristics. In section 3, we present benchmarks demonstrating that our implemenations are viable for use in real code. We then detail the experience of using Typed Racket for this project, both positive and negative. Finally, we discuss other implementations and conclude.

1.1 An Overview of Typed Racket

Typed Racket (Tobin-Hochstadt and Felleisen 2008; Tobin-Hochstadt 2010) is an explicitly typed dialect of Scheme, implemented in Racket (Flatt and PLT 2010). Typed Racket supports both integration with untyped Scheme code as well as a typechecker designed to work with idiomatic Scheme code.

While this paper presents the API of the functional data structures, rather than their implementation, we begin with a brief description of a few key features of the type system.

First, Typed Racket supports explicit polymorphism, which is used extensively in the functional data structure library. Type arguments to polymorphic functions are automatically inferred via *local type inference* (Pierce and Turner 2000). Second, Typed Racket supports untagged rather than disjoint unions. Thus, most data structures presented here are implemented as unions of several distinct structure types.

1.2 Interoperation with External Code

Most Scheme programs are neither purely functional nor typed. That does not prevent them from benefiting from the data structures presented in this paper, however. Typed Racket automatically supports interoperation between typed and untyped code, allowing any program to use the data structures presented here, regardless of whether it is typed. Typed Racket does however enforce its type invariants via software contracts, which can reduce the performance of the structures.

Additionally, using these data structures in no way requires programming in a purely functional style. An mostly-functional Scheme program that does not mutate a list can replace that list with a VList without any problem. Using functional data structures often adds persistence and performance without subtracting functionality.

2. An Introduction to Functional Data Structures

Purely functional data structures, like all data structures, come in many varieties. For this work, we have selected a variety that provide different APIs and performance characteristics. They include several variants of queues, double-ended queues (or deques), priority queues (or heaps), lists, hash lists, tries, red-black trees, and streams. All of the implemented data structures are polymorphic in their element type.

The following subsections describe each of these data structures, many with a number of different implementations with distinct performance characteristics. Each subsection introduces the data structure, specifies its API, provides examples, and then discusses each implementation variant and their performance characteristics.

2.1 Queues

Queues are simple "First In First Out" (FIFO) data structures. We have implemented a wide variety of queues, each with the interface given below. Each queue implementation provides a polymorphic type `(Queue A)`, as well as the following functions:

- *queue* : `(∀ (A) A * → (Queue A))`

  Constructs a queue with the given elements in order. In the `queue` type signature, `∀` is a type constructor used for polymorphic types, binding the given type variables, here `A`. The function type constructor `→` is written infix between arguments and results. The annotation `*` in the function type specifies that `queue` accepts arbitrarily many elements of type `A`, producing a queue of type `(Queue A)`.

- *enqueue* : `(∀ (A) A (Queue A) → (Queue A))`

Inserts the given element (the first argument) into the given queue (the second argument), producing a new queue.

- *head* : `(∀ (A) (Queue A) → A)`

  Returns the first element in the queue. The queue is unchanged.

- *tail* : `(∀ (A) (Queue A) → (Queue A))`

  Removes the first element from the given queue, producing a new queue.

```
> (define que (queue -1 0 1 2 3 4))
> que
- : (Queue Fixnum)
#<Queue>
> (head que)
- : Fixnum
-1
> (head (tail que))
- : Fixnum
0
> (head (enqueue 10 que))
- : Fixnum
-1
```

Banker's Queues The Bankers Queues (Okasaki 1998) are amortized queues obtained using a method of amortization called the Banker's method. The Banker's Queue combines the techniques of lazy evaluation and memoization to obtain good amortized running times. The Bankers Queue implementation internally uses streams (see section 2.4.4) to achieve lazy evaluation. The Banker's Queue provides a amortized running time of $O(1)$ for the operations `head`, `tail` and `enqueue`.

Physicist's Queue The Physicist's Queue (Okasaki 1998) is a amortized queue obtained using a method of amortization called the Physicist's method. The Physicist's Queue also uses the techniques of lazy evaluation and memoization to achieve excellent 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. The Physicist's Queue provides an amortized running time of $O(1)$ for the operations `head`, `tail` and `enqueue`.

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 1998) where lazy components are forced systematically so that no suspension takes more than constant time to execute, assuring ensures good asymptotic worst-case running time for the operations on the data structure. Real-Time Queues have an $O(1)$ worst-case running time for the operations `head`, `tail` and `enqueue`.

Implicit Queue Implicit Queues are a queue data structure implemented by applying a technique called *implicit recursive slowdown* (Okasaki 1998). Implicit recursive slowdown combines laziness with a technique called *recursive slowdown* developed by Kaplan and Tarjan (1995). This technique is simpler than pure recursive slow-down, but with the disadvantage of amortized bounds on the running time. 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 are a queue data structure developed using a bootstrapping technique called *structural decomposition* (Okasaki 1998). In structural decomposition, an implementation that can handle data up to a certain bounded size is used to implement a data structure which can handle data of unbounded size. Bootstrapped Queues give a worst-case running time of $O(1)$ for the operation `head` and $O(\log^* n)$[1] for `tail` and `enqueue`. Our implementation of Bootstrapped Queues uses Real-Time Queues for bootstrapping.

Hood-Melville Queue Hood-Melville Queues are similar to the Real-Time Queues in many ways, but use a different and more complex technique, called *global rebuilding*, to eliminate amortization from the complexity analysis. In global rebuilding, rebalancing is done incrementally, a few steps of rebalancing 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`.

2.2 Deque

Double-ended queues are also known as *deques*. The difference between the queues and the deques lies is that new elements of a deque can be inserted and deleted from either end. We have implemented several deque variants, each discussed below. All the deque data structures implement following interface and have the type `(Deque A)`.

- *deque* : `(∀ (A) A * → (Deque A))`

  Constructs a double ended queue from the given elements in order.

- *enqueue* : `(∀ (A) A (Deque A) → (Deque A))`

  Inserts the given element to the rear of the deque.

- *enqueue-front* : `(∀ (A) A (Deque A) → (Deque A))`

  Inserts the given element to the front of the deque.

- *head* : `(∀ (A) (Deque A) → A)`

  Returns the first element from the front of the deque.

- *last* : `(∀ (A) (Deque A) → A)`

  Returns the first element from the rear of the deque.

- *tail* : `(∀ (A) (Deque A) → (Deque A))`

  Removes the first element from the front of the given deque, producing a new deque.

- *init* : `(∀ (A) (Deque A) → (Deque A))`

  Removes the first element from the rear of the given deque, producing a new deque.

```
> (define dque (deque -1 0 1 2 3 4))
> dque
- : (Deque Fixnum)
#<Deque>
> (head dque)
- : Fixnum
-1
> (last dque)
- : Fixnum
4
> (head (enqueue-front 10 dque))
- : Fixnum
10
> (last (enqueue 20 dque))
- : Fixnum
20
> (head (tail dque))
- : Fixnum
0
```

[1] $\log^* n$ is at most 5 for all feasible queue lengths.

```
> (last (init dque))
- : Fixnum
3
```

Banker's Deque The Banker's Deque is an amortized deque. The Banker's Deque uses the Banker's method and employs the same techniques used in the 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. Implicit Deque provides $O(1)$ amortized running times for the operations `head`, `tail`, `last`, `init`, `enqueue-front` and `enqueue`.

Real-Time Deque The Real-Time Deques 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`.

2.3 Heaps

In order to avoid confusion with FIFO queues, priority queues are also known as *heaps*. A heap is similar to a sortable collection, implemented as a tree, with a comparison function fixed at creation time. There are two requirements that a tree must meet in order for it to be a heap:

- Shape Requirement - All its levels must be full except (possibly) the last level where only rightmost leaves may be missing.
- Parental Dominance Requirement - The key at each node must greater than or equal (max-heap) OR less than or equal (min-heap) to the keys at its children. A tree satisfying this property is said to be *heap-ordered*.

Below, we present several heap variants. Each variant has the type `(Heap A)` and implements the following interface:

- *heap* : `(∀ (A) (A A → Boolean) A * → (Heap A))`
  Constructs a heap from the given elements and comparison function.
- *find-min/max* : `(∀ (A) (Heap A) → A)`
  Returns the min or max element of the given heap.
- *delete-min/max* : `(∀ (A) (Heap A) → (Heap A))`
  Deletes the min or max element of the given heap.
- *insert* : `(∀ (A) A (Heap A) → (Heap A))`
  Inserts an element into the heap.
- *merge* : `(∀ (A) (Heap A) (Heap A) → (Heap A))`
  Merges the two given heaps.

```
> (define hep (heap < 1 2 3 4 5 -1))
> hep
- : (Heap (U Positive-Fixnum Negative-Fixnum))
#<Heap>
> (find-min/max hep)
- : (U Positive-Fixnum Negative-Fixnum)
-1
> (find-min/max (delete-min/max hep))
- : (U Positive-Fixnum Negative-Fixnum)
1
> (define new-hep (heap < -2 3 -4 5))
> (find-min/max (merge hep new-hep))
- : (U Positive-Fixnum Negative-Fixnum)
-4
```

Binomial Heap A Binomial Heap (Vuillemin 1978; Brown 1978) is a heap-ordered binomial tree. Binomial Heaps support a fast `merge` operation using a special tree structure. Binomial Heaps provide a worst-case running time of $O(\log n)$ for the operations `insert`, `find-min/max`, `delete-min/max` and `merge`.

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 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. The Leftist Heaps provide a worst-case running time of $O(\log n)$ for the operations `insert`, `delete-min/max` and `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 which have a very simple implementation and extremely good amortized performance in practice. However, it has proved very difficult to come up with exact asymptotic running time for operations on Pairing Heaps. Pairing Heaps are represented either as a empty heap or a pair of an element and a list of pairing heaps. 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 very similar to balanced binary search trees. The difference between the two is that Splay Heaps do not maintain explicit balance information. Instead, every operation on a splay heap restructures the tree with simple 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 which 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 1998) are similar to pairing heaps as described above, 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 1998) use a technique of bootstrapping called *structural abstraction* (Okasaki 1998), 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. Our implementation of Bootstrapped Heap abstracts over Skew Binomial Heaps.

2.4 Lists

Lists are a fundamental data structure in Scheme. However, while singly-linked lists have the advantages of simplicity and efficiency for some operations, many others are quite expensive. Other data structures can efficiently implement the operations of Scheme's lists, while providing other efficient operations as well. We implement Random Access Lists, Catenable Lists, VLists and Streams. Each implemented variant is explained below. All variants provide the type `(List A)`, and the following interface, which is extended for each implementation:

- *list* : `(∀ (A) A * → (List A))`
  Constructs a list from the given elements, in order.
- *cons* : `(∀ (A) A (List A) → (List A))`
  Adds a given element into the front of a list.
- *first* : `(∀ (A) (List A) → A)`
  Returns the first element of the given list.
- *rest* : `(∀ (A) (List A) → (List A))`
  Produces a new list without the first element.

2.4.1 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:

- *list-ref* : `(∀ (A) (List A) Integer → A)`
  Returns the element at a given location in the list.
- *list-set* : `(∀ (A) (List A) Integer A → (List A))`
  Updates the element at a given location in the list with a new element.

```
> (define lst (list 1 2 3 4 -5 -6))
> lst
- : (U Null-RaList (Root (U Positive-Fixnum
Negative-Fixnum)))
#<Root>
> (first lst)
- : (U Positive-Fixnum Negative-Fixnum)
1
> (first (rest lst))
- : (U Positive-Fixnum Negative-Fixnum)
2
> (list-ref lst 3)
- : (U Positive-Fixnum Negative-Fixnum)
4
> (list-ref (list-set lst 3 20) 3)
- : (U Positive-Fixnum Negative-Fixnum)
20
> (first (cons 50 lst))
- : (U Positive-Fixnum Negative-Fixnum)
50
```

Binary Random Access List Binary Random Access Lists are implemented as using the framework of binary numerical representation using complete binary leaf trees (Okasaki 1998). They have worst-case running times of $O(\log n)$ for the operations `cons`, `first`, `rest`, `list-ref` and `list-set`.

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. Skew Binary Random Access Lists provide worst-case running times of $O(1)$ for the operations `cons`, `head` and `tail` and worst-case running times of $O(\log n)$ for `list-ref` and `list-set` operations.

2.4.2 Catenable List

Catenable Lists are a list data structure with an efficient append operation, achieved using the bootstrapping technique of *structural abstraction* (Okasaki 1998). Catenable Lists are abstracted over Real-Time Queues, and have an amortized running time of $O(1)$ for the basic list operations as well as the following:

- *cons-to-end* : `(∀ (A) A (List A) → (List A))`
  Inserts a given element to the rear end of the list.
- *append* : `(∀ (A) (List A) * → (List A))`
  Appends several lists together.

```
> (define cal (list -1 0 1 2 3 4))
> cal
- : (U EmptyList (List Fixnum))
#<List>
> (first cal)
- : Fixnum
-1
> (first (rest cal))
- : Fixnum
0
> (first (cons 50 cal))
- : Fixnum
50
> (cons-to-end 50 cal)
- : (U EmptyList (List Fixnum))
#<List>
> (define new-cal (list 10 20 30))
> (first (append new-cal cal))
- : Fixnum
10
```

2.4.3 VList

VLists (Bagwell 2002) are a data structure very similar to normal Scheme lists, but with efficient versions of many operations that are much slower on standard lists. VLists combine the extensibility of linked lists with the fast random access capability of arrays. The indexing and length operations of VLists have a worst-case running time of $O(1)$ and $O(\lg n)$ respectively, compared to $O(n)$ for lists. Our VList implementation is built internally on Binary Random Access Lists. VLists provide the standard list API given above, along with many other operations, some of which are given here.

- *last* : `(∀ (A) (List A) → A)`
  Returns the last element of the given list.
- *list-ref* : `(∀ (A) (List A) Integer → A)`
  Gets the element at the given index in the list.

```
> (define vlst (list -1 1 3 4 5))
> vlst
- : (List (U Positive-Fixnum Negative-Fixnum))
#<List>
> (first vlst)
- : (U Positive-Fixnum Negative-Fixnum)
-1
> (first (rest vlst))
- : (U Positive-Fixnum Negative-Fixnum)
1
> (last vlst)
- : (U Positive-Fixnum Negative-Fixnum)
```

```
5
> (length vlst)
- : Integer
5
> (first (cons 50 vlst))
- : (U Positive-Fixnum Negative-Fixnum)
50
> (list-ref vlst 3)
- : (U Positive-Fixnum Negative-Fixnum)
4
> (first (reverse vlst))
- : (U Positive-Fixnum Negative-Fixnum)
5
> (first (map add1 vlst))
- : Integer
0
```

2.4.4 Streams

Streams (Okasaki 1998) are simply lazy lists. They are similar to the ordinary lists and they provide the same functionality and API. Streams are used in many of the foregoing data structures to achieve lazy evaluation. Streams do not change the asymptotic performance of any list operations, but introduce overhead at each suspension. Since streams have distinct evaluation behavior, they are given a distinct type, `(Stream A)`.

2.5 Hash Lists

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

2.6 Tries

A Trie (also known as a Digital Search Tree) is a data structure which takes advantage of the structure of aggregate types to achieve good running times for its operations (Okasaki 1998). 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.

2.7 Red-Black Trees

Red-Black Trees are a classic data structure, consisting of a binary search tree 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 above two invariants together guarantee that the longest possible path with alternating black and red nodes, is no more then twice as long as the shortest possible path, the one with black nodes only. This balancing helps in achieving good running times for the tree operations. Our implementation is based on one by Okasaki (1999). The operations `member?`, `insert` and `delete`, which respectively checks membership, inserts and deletes elements from the tree, have worst-case running time of $O(\log n)$.

3. Benchmarks

To demonstrate the practical usefulness of purely functional data structures, we provide microbenchmarks of a selected set of data structures, compared with both simple implementations based on lists, and imperative implementations. The list based version is implemented in Typed Racket and imperative version is implemented in Racket. The benchmaking was done on a 2.1 GHz Intel Core 2 Duo (Linux) machine and we used Racket version 5.0.0.9 for benchmarking.

In the tables below, all times are CPU time as reported by Racket, including garbage collection time. The times mentioned are in milli seconds and they are time taken for performing each operation 100000 times, averaged over 10 runs. [2]

3.1 Queue Performance

The table in figure 1 shows the performance of the Physicist's Queue, Banker's Queue, Real-Time Queue and Bootstrapped Queue compared with an implementation based on lists, and an imperative queue (Eastlund 2010). [3]

3.2 Heap Performance

The table in figure 2 shows the performance of the Leftist Heap, Pairing Heap, Binomial Heap and Bootstrapped Heap, compared with an implementation based on sorted lists, and a simple imperative heap.

3.3 List Performance

The below table shows the performance of the Skew Binary Random Access List and VList compared with in built lists.

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

4. Experience with Typed Racket

This project involved writing 5300 lines of Typed Racket code, including 1300 lines of tests, almost all written by the first author, who had little previous experience with Typed Racket. This allows us to report on the experience of using Typed Racket for a programmer coming from other languages.

[2] The constructor functions `queue`, `heap` and `list` were repeated only 100 times.

[3] Since 100000 (successive) `tail` (or `dequeue`) operations can not be performed on 1000 element queue, we do not have running time for `tail` operation for for these sizes.

Size	Operation	Physicist's	Banker's	Real-Time	Bootstrapped	List	Imperative
1000	`queue`	16	72	137	20	6	83
	`head`	9	14	30	10	6	54
	`enqueue`	10	127	176	22	256450	73
10000	`queue`	232	887	1576	227	61	746
	`head`	8	17	32	2	7	56
	`enqueue`	11	132	172	18	314710	75
100000	`queue`	3410	13192	20332	2276	860	11647
	`head`	9	16	30	6	8	51
	`tail`	412	312	147	20	7	57
	`enqueue`	12	72	224	18	1289370	84
1000000	`queue`	65590	182858	294310	53032	31480	101383
	`head`	8	17	30	4	7	56
	`tail`	243	1534	1078	20	8	61
	`enqueue`	30	897	1218	20	∞	68

Figure 1. Queue Performance

Size	Operation	Binomial	Leftist	Pairing	Bootstrapped	List	Imperative
1000	`heap`	45	192	30	122	9	306
	`insert`	36	372	24	218	323874	623
	`find`	64	7	6	4	6	8
10000	`heap`	422	2730	340	1283	76	4897
	`insert`	34	358	28	224	409051	628
	`find`	52	9	8	10	7	7
100000	`heap`	6310	40580	4863	24418	1010	69353
	`insert`	33	434	30	198	1087545	631
	`find`	63	8	8	10	7	9
	`delete`	986	528	462	1946	7	439
1000000	`heap`	109380	471588	82840	293788	11140	858661
	`insert`	32	438	28	218	∞	637
	`find`	76	9	6	8	7	7
	`delete`	1488	976	1489	3063	8	812

Figure 2. Heap Performance

4.1 Benefits of Typed Racket

Several features of Typed Racket makes programming in Typed Racket quite enjoyable. First, the type error messages in Typed Racket are very clear and easy to understand. The type checker highlights precise locations which are responsible for type errors. This makes it very easy to debug the type errors.

Second, Typed Racket's syntax is very intuitive, using the infix operator $\rightarrow$ for the type of a function. The Kleene star `*` is used to indicate zero or more elements for rest arguments. $\forall$ is the type constructor used by the polymorphic functions, and so on.

Typed Racket comes with a unit testing framework which makes it simple to write tests, as in the below example:

```
(require typed/test-engine/scheme-tests)
(require "bankers-queue.ss")
(check-expect (head (queue 4 5 2 3)) 4)
(check-expect (tail (queue 4 5 2 3))
                    (queue 5 2 3))
```

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.

4.2 Disadvantages of Typed Racket

Even though overall experience with Typed Racket was positive, there are negative aspects to programming in Typed Racket.

Most significantly for this work, Typed Racket does not support polymorphic non-uniform recursive datatype definitions, which are used extensively by Okasaki (1998). Because of this limitation, many definitions had to be first converted to uniform recursive datatypes before being implemented. For instance, the following definition of `Seq` structure is not allowed by Typed Racket.

```
(define-struct: (A) Seq
  ([elem : A] [recur : (Seq (Pair A A))]))
```

The definition must be converted not to use polymorphic recursion, as follows:

```
(define-struct: (A) Elem ([elem : A]))
(define-struct: (A) Pare
  ([pair : (Pair (EP A) (EP A))]))
(define-type (EP A) (U (Elem A) (Pare A)))
(define-struct: (A) Seq
  ([elem  : (EP A)] [recur : (Seq A)]))
```

Unfortunately, this translation introduces the possibility of illegal states that the typechecker is unable to rule out. We hope to support polymorphic recursion in a future version of Typed Racket.

It is currently not possible to correctly type Scheme functions such as `foldr` and `foldl` because of the limitations of Typed Racket's handling of variable-arity functions (Strickland et al. 2009).

Typed Racket's use of local type inference also leads to potential errors, especially in the presence of precise types for Scheme'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 contains integers, and the second positive integers, neither of which is a subtype of the other. Working around this requires manual annotation to ensure that both vectors have element type `Integer`.

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, as seen in the examples in section 2.

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, as in section 2.4.2. This makes the printing of types confusingly long and difficult to understand, especially in error messages.

5. Comparison with Other Implementations

Our implementations of the presented data structures are very faithful to the original implementations of Purely Functional Data Structures by Okasaki (1998) and VLists and others by Bagwell (2000; 2002). In some cases, we provide additional operations, such as for converting queues to lists.

```
> (queue->list (queue 1 2 3 4 5 6 -4))
- : (Listof (U Positive-Fixnum Negative-Fixnum))
'(1 2 3 4 5 6 -4)
```

We also added an to delete elements from the Red-Black Trees, which was absent in the original implementation. Finally, the heap constructor functions take an explicit comparison function of the type `(A A → Boolean)` as their first argument followed by the elements for the data structure, whereas the original presentation uses ML functors for this purpose. With the above exceptions, the implementation is structurally similar the original work.

We know of no existing comprehensive library of functional data structures for Scheme. Racket's existing collection of user-provided libraries, PLaneT (Matthews 2006), contains an implementation of Random Access Lists (Van Horn 2010), as well as a collection of several functional data structures (Soegaard 2009).

VLists and several other functional data structures have recently been popularized by Clojure (Hickey 2010), a new dialect of Lisp for the Java Virtual Machine.

6. Conclusion

Efficient and productive functional programming requires efficient and expressive functional data structures. In this paper, we present a comprehensive library of functional data structures, implemented and available in Typed Racket. We hope that this enables programmers to write functional programs, and inspires library writers to use functional designs and to produce new libraries to enable functional programming.

Acknowledgments

Thanks to Matthias Felleisen for his support of this work, and to Vincent St-Amour and Carl Eastlund for valuable feedback. Sam Tobin-Hochstadt is supported by a grant from the Mozilla Foundation.

Bibliography

Phil Bagwell. Fast And Space Efficient Trie Searches. Technical report, 2000/334, Ecole Polytechnique Federale de Lausanne, 2000.

Phil Bagwell. Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays. In Implementation of Functional Languages, 14th International Workshop, 2002.

Mark R Brown. Implementation and analysis of binomial queue algorithms. SIAM Journal on Computing, 7(3):298-319, 1978.

Clark Allan Crane. Linear lists and priority queues as balanced binary trees. PhD thesis, Computer Science Department, Stanford University. STAN-CS-72-259., 1972.

Carl Eastlund. Scheme Utilities, version 7. PLaneT Package Repository, 2010.

Matthew Flatt and PLT. Reference: Racket. PLT Scheme Inc., PLT-TR2010-reference-v5.0, 2010.

Michael L. Fredman, Robert Sedgewick, Daniel D. K. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. Algorithmica 1 (1): 111-129, 1986.

Rich Hickey. Clojure. 2010. `http://clojure.org`

Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, 1995.

Jacob Matthews. Component Deployment with PLaneT: You Want it Where? In *Proc. Scheme and Functional Programming*, 2006.

Eugene W. Myers. An applicative random-access stack. *Information Processing Letters* 17(5), pp. 241–248, 1983.

Chris Okasaki. Red-Black Trees in Functional Setting. Journal Functional Programming, 1999.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems* 22(1), pp. 1–44, 2000.

Olin Shivers. SRFI-1: List Library. 1999.

Daniel D. K. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. Journal of the ACM, 32(3):652-686, 1985.

Jens Axel Soegaard. Galore, version 4.2. PLaneT Package Repository, 2009.

T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical Variable-Arity Polymorphism. In *Proc. European Symposium on Programming*, 2009.

Sam Tobin-Hochstadt. Typed Scheme: From Scripts to Programs. PhD dissertation, Northeastern University, 2010.

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. Symposium on Principles of Programming Languages*, 2008.

David Van Horn. RaList: Purely Functional Random-Access Lists, version 2.3. PLaneT Package Repository, 2010.

Jean Vuillemin. A data structure for manipulating priority queues. Communications of the ACM, 21(4):309-315, 1978.