An Overview of Edison

Chris Okasaki *
Department of Computer Science
Columbia University
cdo@cs.columbia.edu

Abstract

Edison is a library of functional data structures implemented in Haskell. It supports three main families of abstractions: sequences, collections (e.g., sets and priority queues), and associative collections (e.g., finite maps). This paper summarizes the design of Edison, with particular attention to how that design is influenced by details of Haskell.

1 Introduction

There is a growing recognition that a useful set of libraries is at least as important to the acceptance of a programming language as the design of the language itself. A library of fundamental data structures such as queues, sets, and finite maps is particularly important in this regard. However, high-quality examples of such libraries, such as the STL [14] in C++ or the the collection classes [3] in Smalltalk, are rare. Edison is a library of efficient data structures suitable for implementation and use in functional programming languages. It is named after Thomas Alva Edison and for the mnemonic value of **EDiSon** (**E**fficient **D**ata **S**trucutres). The current version of the library supports Haskell. Future versions of the library will also support Standard ML and possibly Scheme.

Edison provides several families of abstractions, each with multiple implementations. The main abstractions currently supported by Edison are

- *sequences* (e.g., stacks, queues, deques),
- *collections* (e.g., sets, bags, priority queues where the priority is the element), and
- *associative collections* (e.g., finite maps, priority queues where the priority and element are distinct).

In this paper, I summarize how each of these abstractions is implemented in Haskell, and I discuss how the design of the language influenced the design of the library.

Edison is available through the World Wide Web at

`http://www.cs.columbia.edu/~cdo/edison`

or through the GHC/Hugs CVS repository. In its current state, Edison is mostly a framework. That is, I provide signatures, but not yet a full range of implementations. I hope that Edison can become a community effort, and I welcome anybody to submit new implementations of the Edison abstractions.

*Much of this research was performed in the summer of 1998 at the University of Glasgow, with funds from the Scottish Higher Education Development Council.

```
data Maybe2 a b = Just2 a b | Nothing2      deriving (Eq,Show)
data Maybe3 a b c = Just3 a b c | Nothing3  deriving (Eq,SHow)

class Eq a => Hash a where
  hash :: a -> Int
  -- forall x,y :: a. (x == y) implies (hash x == hash y)

class Hash a => UniqueHash a
  -- no new methods, just a stronger invariant
  -- forall x,y :: a. (x == y) iff (hash x == hash y)

class UniqueHash a => ReversibleHash a where
  unhash :: Int -> a
  -- forall x :: a. unhash (hash x) == x
```

Figure 1: The `EdisonPrelude` module.

```
module BreadthFirst where

import EdisonPrelude
import qualified SimpleQueue as Q

data Tree a = Empty | Node a (Tree a) (Tree a)

breadthFirst :: Tree a -> [a]
breadthFirst t = bfs (Q.single t)
  where bfs q =
          case Q.lview q of
            Just2 (Node x l r) q' -> x : bfs (Q.snoc (Q.snoc q' l) r)
            Just2 Empty q' -> bfs q'
            Nothing2 -> []
```

Figure 2: Sample program using Edison.

2 General Organization

Each family of abstractions is implemented as a class hierarchy and each data structure is implemented as a Haskell module. For the operations in each class and module, I have attempted to choose names that are as standard as possible. This means that operations for different abstractions frequently share the same name (`empty`, `null`, `size`, etc.). It also means that in many cases I have reused names from the Prelude. Therefore, Edison modules should nearly always be imported `qualified`. The one Edison module that is typically imported unqualified is the `EdisonPrelude`, shown in Figure 1, which defines a few utility types in the `Maybe` family used by every other Edison module, as well as a few classes related to hashing.

When importing Edison modules, I recommend renaming each module using the `as` keyword. See, for example, the sample program in Figure 2, where the imported module `SimpleQueue` has been renamed locally as `Q`. This both reduces the overhead of qualified names and makes substituting one module for another as painless as possible. If I wanted to replace `SimpleQueue` with a fancier implementation such as `HoodMelvilleQueue`, I could do so merely by modifying the import line. Such substitutions are further facilitated by the convention that related data structures should use the same type name. For example, most implementations of sequences define a type constructor named `Seq`.

The sample program in Figure 2 also illustrates another important point about Edison—although each abstraction is defined in terms of type classes, all the operations on each data structure are also available directly from the data structure's module. If we wanted to access methods such as `single` through the type class instead, we could change the line

```
import qualified SimpleQueue as Q
```

to

```
import qualified Sequence as Q  -- import class
import SimpleQueue (Seq)        -- import instance
```

and then use a type annotation somewhere inside the `breadthFirst` function to indicate that the intermediate queues are of type `Seq a`. Note that, because I am selectively importing only the type constructor `Seq` from `SimpleQueue`, I do not bother importing it `qualified`.

3 Sequences

The *sequence* abstraction is usually viewed as a hierarchy of ADTs including lists, queues, deques, catenable lists, etc. However, such a hierarchy is based on efficiency rather than functionality. For example, a list supports all the operations that a deque supports, even though some of the operations may be inefficient. Hence, in Edison, all sequence data structures are defined as instances of the single `Sequence` class:

```
class (Functor s, MonadPlus s) => Sequence s
```

As expressed by the context, all sequences are also instances of `Functor`, `Monad`, and `MonadPlus`. In addition, all sequences are expected to be instances of `Eq` and `Show`, although this is not enforceable in Haskell.[1] Figure 3 summarizes all the methods on sequences.

Sequences are currently the most populated abstraction in Edison. There are six basic implementations of sequences, including ordinary lists, join lists, simple queues [1], banker's queues [9], random-access stacks [6], random-access lists [7], Braun trees [4, 8], and binary random-access lists [9], plus two *sequence adaptors*, which are representations of sequences parameterized by other representations of sequences. One adds an explicit size field to an existing implementation of sequences and the other reverses the orientation of an existing implementation of sequences so that adding an element to the left of the sequence actually adds the element to the right of the underlying sequence.

4 Collections

The *collection* abstraction includes sets, bags, and priority queues (heaps). Collections are defined in Edison as a set of eight classes, organized in the hierarchy shown in Figure 4. These classes make essential use of multi-parameter type classes, as in [11]. All collections assume at least an equality relation on elements, and many also assume an ordering relation. The use of multi-parameter type classes allows any particular instance to assume further properties as necessary (such as hashability).

The hierarchy contains a root class, `CollX`, together with seven subclasses satisfying one or more of three orthogonal sub-properties:

- *Uniqueness.* Each element in the collection is unique (i.e., no two elements in the collection are equal). These subclasses, indicated by the name `Set`, represent sets rather than bags.

[1]Enforcing this condition would require being able to write constraints like $\forall a.$`Eq` a `=> Eq (s` a`)` inside class contexts.

Sequence Methods

Constructors:
`empty`, `single`, `cons`, `snoc`, `append`, `fromList`, `copy`, `tabulate`

Destructors:
`lview`, `lhead`, `ltail`, `rview`, `rhead`, `rtail`

Observers:
`null`, `size`, `toList`

Concat and reverse:
`concat`, `reverse`, `reverseOnto`

Maps and folds:
`map`, `concatMap`, `foldr`, `foldl`, `foldr1`, `foldl1`, `reducer`, `reducel`, `reduce1`

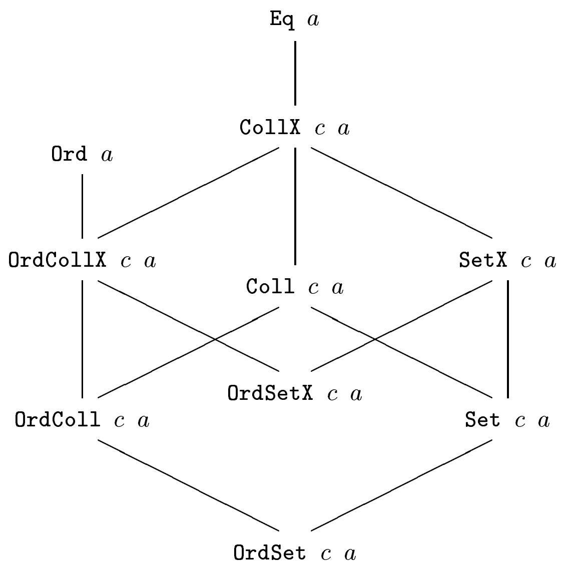
Subsequences:
`take`, `drop`, `splitAt`, `subseq`

Predicate-based operations:
`filter`, `partition`, `takeWhile`, `dropWhile`, `splitWhile`

Index-based operations:
`inBounds`, `lookup`, `lookupM`, `lookupWithDefault`, `update`, `adjust`,
`mapWithIndex`, `foldrWithIndex`, `foldlWithIndex`

Zips and unzips:
`zip`, `zip3`, `zipWith`, `zipWith3`, `unzip`, `unzip3`, `unzipWith`, `unzipWith3`

Figure 3: Summary of methods for the `Sequence` class.



CollX	OrdCollX	SetX	OrdSetX
`empty,insert`	`deleteMin`	`intersect`	*no methods*
`union,delete`	`unsafeInsertMin`	`difference`	
`null,size`	`filterLT`	`subset`	
`member,count`	...	`subsetEq`	
...			

Coll	OrdColl	Set	OrdSet
`toSeq`	`minElem`	`insertWith`	*no methods*
`lookup`	`foldr,foldl`	`unionWith`	
`fold`	`toOrdSeq`	`intersectWith`	
`filter`	...	...	
...			

Figure 4: The collection class hierarchy, with typical methods for each class.

Collection Methods

Constructors:

`CollX`: `empty`, `single`, `insert`, `insertSeq`, `union`, `unionSeq`, `fromSeq`
`OrdCollX`: `unsafeInsertMin`, `unsafeInsertMax`, `unsafeFromOrdSeq`, `unsafeAppend`
`Set`: `insertWith`, `insertSeqWith`, `unionl`, `unionr`, `unionWith`, `unionSeqWith`, `fromSeqWith`

Destructors:

`OrdColl`: `minView`, `minElem`, `maxView`, `maxElem`

Deletions:

`CollX`: `delete`, `deleteAll`, `deleteSeq`
`OrdCollX`: `deleteMin`, `deleteMax`

Observers:

`CollX`: `null`, `size`, `member`, `count`
`Coll`: `lookup`, `lookupM`, `lookupAll`, `lookupWithDefault`, `toSeq`
`OrdColl`: `toOrdSeq`

Filters and partitions:

`OrdCollX`: `filterLT`, `filterLE`, `filterGT`, `filterGE`,
`partitionLT_GE`, `partitionLE_GT`, `partitionLT_GT`
`Coll`: `filter`, `partition`

Set operations:

`SetX`: `intersect`, `difference`, `subset`, `subsetEq`
`Set`: `intersectWith`

Folds:

`Coll`: `fold`, `fold1`
`OrdColl`: `foldr`, `foldl`, `foldr1`, `foldl1`

Figure 5: Summary of methods for the collection classes.

- *Ordering.* The elements have a total ordering and it is possible to process the elements in non-decreasing order. These subclasses, indicated by the `Ord` prefix, typically represent either priority queues (heaps) or sets/bags implemented as binary search trees.
- *Observability.* An observable collection is one in which it is possible to view the elements in the collection. The `X` suffix indicates lack of observability. This property is discussed in greater detail below.

Figure 5 summarizes all the methods on collections. Note that neither `OrdSetX` nor `OrdSet` add any new methods, which is why there is no explicit dependency between these classes in the hierarchy. These classes serve as mere abbreviations for the combinations of `OrdCollX/SetX` and `OrdColl/Set`, respectively.

As with sequences, the hierarchy of collections is determined by functionality rather than efficiency. For example, the `member` function is included in the root class of the hierarchy even though it is inefficient for many implementations, such as heaps.

Because collections encompass a wide range of abstractions, there is no single name that is suitable for all collection type constructors. However, most modules implementing collections will define a type constructor named either `Bag`, `Set`, or `Heap`.

Edison currently supports one implementation of sets (unbalanced binary search trees), four implementations of heaps (leftist heaps [5], skew heaps [13], splay heaps [9], and lazy pairing heaps [9]), and one heap adaptor that maintains the minimum element of a heap separate from the rest of the heap. This heap adaptor is particularly useful in conjunction with splay heaps.

4.1 Observability

Note that the equality relation defined by the `Eq` class is not necessarily true equality. Very often it is merely an equivalence relation, where equivalent values may be distinguishable by other means. For example, we might consider two binary search trees to be equal if they contain the same elements, even if their shapes are different.

Because of this phenomenon, implementations of observable collections (i.e., collections where it is possible to inspect the elements) are rather constrained. Such an implementation must retain the actual elements that were inserted. For example, it is not possible in general to represent an observable bag as a finite map from elements to counts, because even if we know that a given bag contains, say, three elements from some equivalence class, we do not necessarily know *which* three.

On the other hand, implementations of *non-observable* collections have much greater freedom to choose abstract representations of each equivalence class. For example, representing a bag as a finite map from elements to counts works fine if we never need to know *which* representatives from an equivalence class are actually present. As another example, consider the `UniqueHash` class defined in the Edison Prelude. If we know that the `hash` function yields a unique integer for each equivalence class, then we can represent a collection of hashable elements simply as a collection of integers. With such a representation, we can still do many useful things like testing for membership—we just can't support functions like `fold` or `filter` that require the elements themselves, rather than the hashed values.[2]

4.2 Unsafe Operations

Ordered collections support a number of operations with names like `unsafeInsertMin` and `unsafeFromOrdSeq`. These are important special cases with preconditions that are too expensive to check at runtime. For example, `unsafeFromOrdSeq` converts a sorted sequence of elements into a collection. In contrast to `fromSeq`, which converts an unsorted sequence into a collection, `unsafeFromOrdSeq` can be implemented particularly efficiently for data structures like binary search trees. The behavior of these operations is undefined if the preconditions are not satisfied, so the `unsafe` prefix is intended to remind the programmer that these operations are accompanied by a proof obligation.

The one place where I have violated this convention is in the `Set` class, where there is a whole family of operations with names like `insertWith` and `unionWith`. These functions take a combining function that is used to resolve collisions. For example, when inserting an element into a set that already contains that element, the combining function is called on the new and old elements to determine which element will remain in the new set.[3] The combining functions typically return one element or the other, but they can also combine the elements in non-trivial ways. These combining functions are required to satisfy the precondition that, given two equal elements, they return a third element that is equal to the other two.

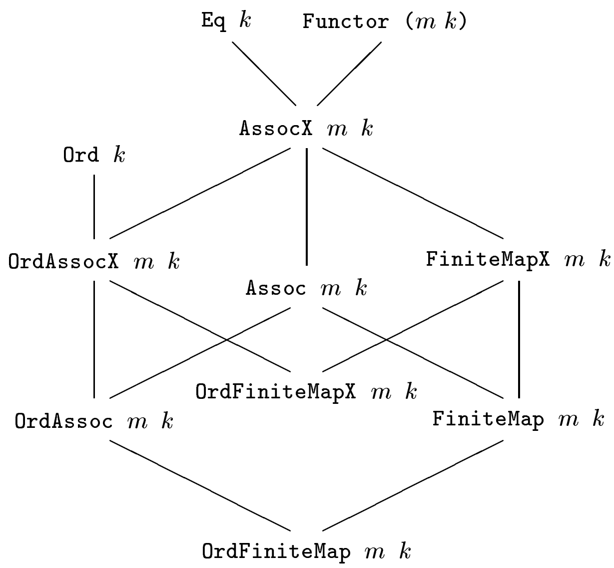
5 Associative Collections

The *associative-collection* abstraction includes finite maps, finite relations, and priority queues where the priority is distinct from the element. Associative collections are defined in Edison as a set of eight classes, organized in the hierarchy shown in Figure 6. Notice that this hierarchy mirrors the hierarchy for collections, but with the addition of `Functor` as a superclass of every associative collection. Like collections, associative collections depend heavily on multi-parameter type classes.

---

[2] In fact, we can even support `fold` and `filter` if the hashing function is *reversible*, but this is relatively uncommon.

[3] Such a combining function is useful only when nominally equal elements are distinguishable in other ways—that is, when the "equality" relation is really an equivalence relation. However, this is extremely common.



AssocX	OrdAssocX	FiniteMapX	OrdFiniteMapX
`empty,insert`	`minElem`	`insertWith`	*no methods*
`union,delete`	`deleteMin`	`unionWith`	
`null,size`	`unsafeInsertMin`	`intersectWith`	
`lookup`	`foldr,foldl`	`difference`	
`map,fold`	`filterLT`	`subset`	
`filter`	...	...	
...			

Assoc	OrdAssoc	FiniteMap	OrdFiniteMap
`toSeq`	`minElemWithKey`	`unionWithKey`	*no methods*
`mapWithKey`	`foldrWithKey`	`intersectWithKey`	
`foldWithKey`	`toOrdSeq`	...	
`filterWithKey`	...		
...			

Figure 6: The associative-collection class hierarchy, with typical methods for each class.

The operations on associative collections are similar to the operations on collections. The differences arise from having a separate key and element, rather than just an element. One significant implication of this separation is that many of the methods move up in the hierarchy, because elements are always observable for associative collections even though keys may not be. Figure 7 summarizes all the methods on associative collections.

Edison currently supports two implementations of finite maps (association lists and Patricia trees [10]).

Because collections and associative collections are so similar, it is tempting to merge them into one class hierarchy, either by defining collections to be associative collections whose elements are of the unit type or by defining associative collections to be collections whose elements are pairs of type `Association k a`, where the ordering on associations is inherited from the keys only. For example, Peyton Jones [11] follows this latter approach. Edison rejects both approaches, however, because both carry unacceptable performance penalties. The former requires extra space for the unnecessary unit values and the latter injects at least one extra level of indirection into every key access. The implementor is free to define any particular implementation of a data structure in one of these ways, trading a small performance penalty for reduced development costs, but it would be wrong for the design of the library to mandate that *every* implementation of an abstraction pay these penalties.

6 Testing

Each abstraction in Edison has an associated test suite implemented under QuickCheck [2]. To support both this testing and any testing of applications built on top of Edison, every Edison data structure is defined to be an instance of the `Arbitrary` class. This class is used by QuickCheck to generate random versions of each data structure, which are then passed to the routines that check the desired invariants, such as

```
cons x xs == append (single x) xs
```

The QuickCheck test suite is a relatively new addition to Edison. Compared to the old test suite, I estimate that the QuickCheck test suite took less that 25% of the effort to develop, and provides much better coverage as well! I highly recommend using QuickCheck in any application with a relatively well-understood specification.

7 Commentary

There are many places where the design of Haskell has influenced the design of Edison in non-obvious ways. In addition, there are several places where Edison runs up against limits in the design of Haskell.

7.1 Fixity

Because qualified infix symbols are fairly ugly, Edison avoids infix symbols as much as possible. For example, the sequence catenation function is named `append` instead of `++`.

7.2 Error handling

Because Haskell has no good way to recover from errors, Edison avoids signalling errors if there is any reasonable alternative. For many functions, it is easy to avoid an error by returning the `Maybe` type (or something similar), but sometimes, as with the `head` function on lists and the corresponding `lhead` function on sequences, this approach is just too painful. For `lhead` of an empty sequence, there really is no choice but to signal an error, but other times there is a reasonable alternative. For example, Edison defines both `ltail` of the empty sequence and `take` of a negative argument to return the empty sequence even though the corresponding Prelude functions would signal errors in both cases.

Associative-Collection Methods

Constructors:

`AssocX`: `empty`, `single`, `insert`, `insertSeq`, `union`, `unionSeq`, `fromSeq`
`OrdAssocX`: `unsafeInsertMin`, `unsafeInsertMax`, `unsafeFromOrdSeq`, `unsafeAppend`
`FiniteMapX`: `insertWith`, `insertWithKey`, `insertSeqWith`, `insertSeqWithKey`,
`unionl`, `unionr`, `unionWith`, `unionSeqWith`, `fromSeqWith`, `fromSeqWithKey`
`FiniteMap`: `unionWithKey`, `unionSeqWithKey`

Destructors:

`OrdAssocX`: `minView`, `minElem`, `maxView`, `maxElem`
`OrdAssoc`: `minViewWithKey`, `minElemWithKey`, `maxViewWithKey`, `maxElemWithKey`

Deletions:

`AssocX`: `delete`, `deleteAll`, `deleteSeq`
`OrdAssocX`: `deleteMin`, `deleteMax`

Observers:

`AssocX`: `null`, `size`, `member`, `count`, `lookup`, `lookupM`, `lookupAll`, `lookupWithDefault`, `elements`
`Assoc`: `toSeq`, `keys`
`OrdAssoc`: `toOrdSeq`

Modifiers:

`AssocX`: `adjust`, `adjustAll`

Maps and folds:

`AssocX`: `map`, `fold`, `fold1`
`OrdAssocX`: `foldr`, `foldl`, `foldr1`, `foldl1`
`Assoc`: `mapWithKey`, `foldWithKey`
`OrdAssoc`: `foldrWithKey`, `foldlWithKey`

Filters and partitions:

`AssocX`: `filter`, `partition`
`OrdAssocX`: `filterLT`, `filterLE`, `filterGT`, `filterGE`,
`partitionLT_GE`, `partitionLE_GT`, `partitionLT_GT`
`Assoc`: `filterWithKey`, `partitionWithKey`

Set-like operations:

`FiniteMapX`: `intersectWith`, `difference`, `subset`, `subsetEq`
`FiniteMap`: `intersectWithKey`

Figure 7: Summary of methods for the associative-collection classes.

7.3 Map

It may be surprising that the collection hierarchy does not include a `map` method. In fact, Edison includes a utility function

```
map :: (Coll cin a, CollX cout b) => (a -> b) -> (cin a -> cout b)
map f xs = fold (insert . f) empty xs
```

but this function is not a method, so there is no hope of substituting something more efficient for a particular implementation of collections. But how could this operation be implemented more efficiently? For example, it is tempting to implement `map` on a binary search tree by the usual map function for trees. However, besides limiting `map` to the special case where `cin` and `cout` are identical, this implementation is incorrect. There is no guarantee that `f` preserves the ordering of elements, so the result would not in general be a valid binary search tree. Many Edison data structures can and do support a function `unsafeMapMonotonic` that assumes that `f` preserves ordering, leaving this fact as a proof obligation for the user, but this function is not general enough to deserve to be a method.

7.4 Defaults

Haskell supports default implementations of methods, but Edison makes almost no use of this language feature. The difficulty is that there is very often more than one implementation that could play this role. For example, consider the `insertSeq` method for inserting a sequence of elements into a collection. There are at least two equally good "default" implementations of this method: the first inserts each element of the sequence into the collection, and the second converts the sequence into a collection and then unions this new collection with the old one. Arbitrarily designating one of these implementations as *the* default would simply lead to performance bugs in which the implementor forgets to overide the default method, thinking that the other implementation has been chosen as the default.

The solution in Edison is to provide, for each family of abstractions, a separate module containing all these myriad default implementations, with names like `insertSeqUsingFoldr` and `insertSeqUsingUnion`. Then, each data structure module contains a set of definitions of the form

```
insertSeq = insertSeqUsingFoldr
```

for those methods for which a default implementation is appropriate.

7.5 Limitations on Contexts

Haskell's restrictions on the form of type contexts occasionally prove too restrictive. For example, the root of the associative-collection class hierarchy is defined as

```
class (Eq k, Functor (m k)) => AssocX m k
```

but the `(m k)` in the `Functor` context is not allowed — at least, not in Haskell 98. An unsatisfying workaround is to simply delete the `Functor` part of the context and add a `map` method to `AssocX`.

Similarly, it would be useful to be able to define collections based on hashing, as in

```
newtype HashColl c a = H (c Int)

instance (UniqueHash a, CollX c Int) => CollX (HashColl c) a where
  single = single . hash
  ...
```

but the `Int` in the `CollX c Int` context is not allowed.

For further discussion of Haskell's limitations on contexts, see [12].

8 Final Words

Haskell programmers, indeed functional programmers in general, too often reach for lists when an ADT would be more appropriate. Without Edison or some similar library, I fear this trend will continue indefinitely.

A library like Edison will only be successful if it is embraced by the community. I welcome community involvement at every level from design to implementation. I am especially eager for user feedback, and I repeat my earlier invitation for anybody to submit new implementations of the Edison abstractions.

Acknowledgements

Thanks to Simon Peyton Jones for many discussions about the design of Edison. Thanks also Ralf Hinze and Sigbjorne Finne, who have each contributed to the Edison infrastructure. Finally, thanks to Koen Claessen and John Hughes for their wonderful QuickCheck tool.

References

[1] F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982.

[2] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN International Conference on Functional Programming*, September 2000.

[3] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, October 1992.

[4] Rob R. Hoogerwoord. A logarithmic implementation of flexible arrays. In *Conference on Mathematics of Program Construction*, volume 669 of *LNCS*, pages 191–207. Springer-Verlag, July 1992.

[5] Donald E. Knuth. *Searching and Sorting*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[6] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–248, December 1983.

[7] Chris Okasaki. Purely functional random-access lists. In *Conference on Functional Programming Languages and Computer Architecture*, pages 86–95, June 1995.

[8] Chris Okasaki. Three algorithms on Braun trees. *Journal of Functional Programming*, 7(6):661–666, November 1997.

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

[10] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, September 1998.

[11] Simon Peyton Jones. Bulk types with class. In *Glasgow Workshop on Functional Programming*, July 1996.

[12] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[13] Daniel D. K. Sleator and Robert E. Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, February 1986.

[14] Alexander Stepanov and Meng Lee. The standard template library. Technical report, Hewlett-Packard, 1995.