Electronic Notes in Theoretical Computer Science 41 No. 1 (2001)

URL: <http://www.elsevier.nl/locate/entcs/volume41.html> 14 pages

An Overview of Edison

Chris Okasaki 1*,*2

*Department of Computer Science Columbia University*

*New York, New York, USA*

**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.

# 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 **ED**i**S**on (**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 imple- mentations. 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 ele- ment), and

1 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.

2 Email: [cdo@cs.columbia.edu](mailto:cdo@cs.columbia.edu)

*◯*c *2001 Published by Elsevier Science B. V.* Open access under [CC BY NC ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

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

Fig. 1. The EdisonPrelude module.

* *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.

# 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

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 -> []

Fig. 2. Sample program using Edison.

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.

# 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

**Constructors:**

# Sequence Methods

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

Fig. 3. Summary of methods for the Sequence class.

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 in- stances of Eq and Show, although this is not enforceable in Haskell. 3 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 rep- resentations of sequences. One adds an explicit size field to an existing imple- mentation of sequences and the other reverses the orientation of an existing

3 Enforcing this condition would require being able to write constraints like *∀a*.Eq *a* => Eq (s *a*) inside class contexts.

implementation of sequences so that adding an element to the left of the se- quence actually adds the element to the right of the underlying sequence.

# 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 hier- archy 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 prop- erties 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.
* *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 depen- dency between these classes in the hierarchy. These classes serve as mere abbreviations for the combinations of OrdCollX/SetX and OrdColl/Set, re- spectively.

As with sequences, the hierarchy of collections is determined by function- ality rather than efficiency. For example, the member function is included in the root class of the hierarchy even though it is inefficient for many implemen- tations, 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 main- tains the minimum element of a heap separate from the rest of the heap. This heap adaptor is particularly useful in conjunction with splay heaps.

Eq *a*

CollX *c a*

Ord *a*

✟✟✟

✟✟✟

❍❍❍

❍❍❍

✟

OrdCollX *c a*

❍❍❍

Coll *c a*

❍

SetX *c a*

✟✟✟

❍❍✟✟✟ ❍❍❍✟✟

✟✟✟ ❍❍ ✟✟ ❍❍❍

✟

OrdColl *c a*

❍❍❍

OrdSetX *c a*

❍

Set *c a*

✟✟✟

❍❍ ✟✟

❍

✟

* + ✟

OrdSet *c a*

|  |  |  |  |
| --- | --- | --- | --- |
| CollX | OrdCollX | SetX | OrdSetX |
| empty,insert  union,delete | deleteMin  unsafeInsertMin | intersect  difference | *no methods* |
| null,size | filterLT | subset |  |
| member,count | *···* | subsetEq |  |
| *···* |  |  |  |
| Coll | OrdColl | Set | OrdSet |
| toSeq  lookup | minElem  foldr,foldl | insertWith  unionWith | *no methods* |
| fold | toOrdSeq | intersectWith |  |
| filter | *···* | *···* |  |
| *···* |  |  |  |

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

* 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.

# 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

Fig. 5. Summary of methods for the collection classes.

Because of this phenomenon, implementations of observable collections (i.e., collections where it is possible to inspect the elements) are rather con- strained. 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 repre- sentation, 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. 4

* 1. *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. 5 The com- bining functions typically return one element or the other, but they can also combine the elements in non-trivial ways. These combining functions are re- quired to satisfy the precondition that, given two equal elements, they return a third element that is equal to the other two.

# 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.

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

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

5 Such a combining function is useful only when nominally equal elements are distinguish-

able in other ways—that is, when the “equality” relation is really an equivalence relation. However, this is extremely common.

Eq *k* Functor (*m k*)

❅❅ //

❅ /

AssocX *m k*

Ord *k*

✟✟✟

✟✟✟

❍❍❍

❍❍❍

✟

OrdAssocX *m k*

❍❍❍

Assoc *m k*

❍

FiniteMapX *m k*

✟✟✟

❍❍✟✟✟ ❍❍❍✟✟

✟✟✟ ❍❍ ✟✟ ❍❍❍

✟ OrdFiniteMapX *m k* ❍

OrdAssoc *m k*

❍❍❍

❍❍

❍

❍

FiniteMap *m k*

✟✟✟

✟✟

✟

✟

OrdFiniteMap *m k*

|  |  |  |  |
| --- | --- | --- | --- |
| AssocX | OrdAssocX | FiniteMapX | OrdFiniteMapX |
| empty,insert  union,delete | minElem  deleteMin | insertWith  unionWith | *no methods* |
| null,size | unsafeInsertMin | intersectWith |  |
| lookup | foldr,foldl | difference |  |
| map,fold | filterLT | subset |  |
| filter | *···* | *···* |  |
| *···* |  |  |  |
| Assoc | OrdAssoc | FiniteMap | OrdFiniteMap |
| toSeq  mapWithKey | minElemWithKey  foldrWithKey | unionWithKey  intersectWithKey | *no methods* |
| foldWithKey | toOrdSeq | *···* |  |
| filterWithKey | *···* |  |  |
| *···* |  |  |  |

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

# 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

Fig. 7. Summary of methods for the associative-collection classes.

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 as- sociative collections whose elements are of the unit type or by defining associa- tive 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 im- plementor is free to define any particular implementation of a data structure in one of these ways, trading a small performance penalty for reduced devel- opment costs, but it would be wrong for the design of the library to mandate that *every* implementation of an abstraction pay these penalties.

# 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. Com- pared 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.

# 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.

* 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 ++.

* 1. *Error handling*

Because Haskell has no good way to recover from errors, Edison avoids sig- nalling errors if there is any reasonable alternative. For many functions, it is easy to avoid an error by returning the Maybe type (or something simi- lar), 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.

* 1. *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 tempt- ing 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, leav- ing this fact as a proof obligation for the user, but this function is not general enough to deserve to be a method.

* 1. *Defaults*

Haskell supports default implementations of methods, but Edison makes al- most 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 sec- ond 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 for- gets 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.

* 1. *Limitations on Contexts*

Haskell’s restrictions on the form of type contexts occasionally prove too re- strictive. 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 hash- ing, 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].

# 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 com- munity. I welcome community involvement at every level from design to im- plementation. I am especially eager for user feedback, and I repeat my earlier invitation for anybody to submit new implementations of the Edison abstrac- tions.

# Acknowledgements

Thanks to Simon Peyton Jones for many discussions about the design of Edi- son. 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. Burton, F. W., *An efficient functional implementation of FIFO queues*, Information Processing Letters **14** (1982), pp. 205–206.
2. Claessen, K. and J. Hughes, *Quickcheck: A lightweight tool for random testing of haskell programs*, in: *ACM SIGPLAN International Conference on Functional Programming*, 2000.
3. Cook, W. R., *Interfaces and speciﬁcations for the Smalltalk-80 collection classes*, in: *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1992, pp. 1–15.
4. Hoogerwoord, R. R., *A logarithmic implementation of flexible arrays*, in: *Conference on Mathematics of Program Construction*, LNCS **669** (1992), pp. 191–207.
5. Knuth, D. E., “Searching and Sorting,” The Art of Computer Programming **3**, Addison-Wesley, 1973, 150–152 pp.
6. Myers, E. W., *An applicative random-access stack*, Information Processing Letters **17** (1983), pp. 241–248.
7. Okasaki, C., *Purely functional random-access lists*, in: *Conference on Functional Programming Languages and Computer Architecture*, 1995, pp. 86–95.
8. Okasaki, C., *Three algorithms on Braun trees*, Journal of Functional Programming **7** (1997), pp. 661–666.
9. Okasaki, C., “Purely Functional Data Structures,” Cambridge University Press, 1998.
10. Okasaki, C. and A. Gill, *Fast mergeable integer maps*, in: *Workshop on ML*, 1998, pp. 77–86.
11. Peyton Jones, S., *Bulk types with class*, in: *Glasgow Workshop on Functional Programming*, 1996.
12. Peyton Jones, S., M. Jones and E. Meijer, *Type classes: an exploration of the design space*, in: *Haskell Workshop*, 1997.
13. Sleator, D. D. K. and R. E. Tarjan, *Self-adjusting heaps*, SIAM Journal on Computing **15** (1986), pp. 52–69.
14. Stepanov, A. and M. Lee, *The standard template library*, Technical report, Hewlett-Packard (1995).