Another iteration on "A synthesis of several sorting algorithms"

Brian T. Howard
Department of Computing and Information Sciences
Kansas State University
Manhattan, Kansas 66506 USA
bhoward@cis.ksu.edu

October 18, 1994

Abstract

In "A synthesis of several sorting algorithms", Darlington showed how to use program transformation techniques to develop versions of six well-known sorting algorithms. We provide more evidence for the naturalness of the resulting taxonomy of algorithms by showing how it follows almost immediately from a consideration of the types of the objects involved. By exploiting the natural operations of iteration and coiteration over recursively defined data types, we may automatically derive the structure of each algorithm.

1 Introduction

In "A synthesis of several sorting algorithms" [5], Darlington showed how to use program transformation techniques to develop versions of six well-known sorting algorithms. By systematically constructing these algorithms from a common high-level specification he wished to expose the underlying relationships among the algorithms and provide an explanation for their origin. The resulting "family tree" of algorithms was subsequently modified by Barstow [3] and Merritt [20] to reveal more symmetry among the derivations, providing an elegant taxonomy for this collection of sorting routines.

The purpose of this note is to provide more evidence for the naturalness of this modified taxonomy by showing how it follows almost immediately from a consideration of the types of the objects involved. This work is an outgrowth of current research in the Programming Language Semantics group at Kansas State University on applications of type theory and realizability to problems of practical algorithm design; it reflects our view that significant portions of the process of programming may profitably be guided by attention to the logical structure of programs and data.

The following table shows the classification of the Insertion Sort, Selection Sort, Merge Sort, and Quick Sort algorithms given by Barstow and Merritt:

	Singleton	Equal-size
Easy Split/Hard Join	Insertion Sort	Merge Sort
Hard Split/Easy Join	Selection Sort	Quick Sort

All four algorithms are instances of the Divide-and-Conquer strategy. The columns of the table correspond to whether the subproblems are taken to be of roughly equal size or whether a single element is split off as a (trivially solvable) subproblem. The rows of the table are determined by whether a significant effort is put into splitting the original problem into subproblems or whether the main work is performed in joining the solutions of the subproblems back into the final result. Barstow refers to these as "Split by Value" and "Split by Position," respectively, since we are distinguishing whether the values need to be examined in doing the split or whether we may split simply by choosing a fixed location in the list.

The remaining two sorting algorithms synthesized by Darlington are what he refers to as Exchange Sort and Bubble Sort. The former, which more closely matches what is traditionally called Bubble Sort, is identified by Barstow as an instance of Selection Sort (indeed, Knuth [17] gives Exchange Selection Sort as an alternate name for Bubble Sort), where the auxiliary function of selecting the minimum element is performed by a pass of in-place exchange operations on adjacent elements. Barstow observes that the dual of Bubble Sort, which Knuth calls Straight Insertion Sort or Sinking Sort, is the corresponding instance of Insertion Sort where the insertion is done by in-place exchanges. Darlington's Bubble Sort, by contrast, is merely a slower version of his Exchange Sort, where at most one exchange is performed on each pass through the list. We will not be considering these versions of sorting further in this note.

2 Types, iteration, and coiteration

If we wish to assign types to the data structures involved in these algorithms, then the choice between splitting off a singleton and splitting into two equal-size subproblems neatly corresponds to the traditional choice of whether to represent a list as a Cons-list or an Append-list, i.e., whether the fundamental operation on lists is to cons an element onto the head of a list or to append two lists tail to head. In Standard ML we might declare these types for lists of integers as follows (but see below):

```
datatype ConsList = Nil
                  | Cons of int * ConsList
datatype AppendList = Empty
                    | Singleton of int
                    | Append of AppendList * AppendList
```

We may describe these two types somewhat more traditionally, and certainly more compactly (at the cost of omitting the constructor names), by giving the recursive type equations

$$\begin{aligned} \texttt{ConsList} &= 1 + (\texttt{int} \times \texttt{ConsList}) \\ \texttt{AppendList} &= 1 + \texttt{int} + (\texttt{AppendList} \times \texttt{AppendList}), \end{aligned}$$

where 1 is a one-element type (called `unit` in ML), $\times$ is the cartesian product (corresponding to the operator `*` in ML), and $+$ is the disjoint union (corresponding to the `|`). If we write $C(\alpha)$ for the parameterized type expression $1 + (\texttt{int} \times \alpha)$, and similarly write $A(\alpha)$ for $1 + \texttt{int} + (\alpha \times \alpha)$, then these equations become $\texttt{ConsList} = C(\texttt{ConsList})$ and $\texttt{AppendList} = A(\texttt{AppendList})$, i.e., `ConsList` and `AppendList` are fixed points of C and A, respectively. Taking this view, a more accurate representation of the types of Cons-lists and Append-lists in ML is

```
datatype 'a C = Nil
              | Cons of int * 'a
datatype 'a A = Empty
              | Singleton of int
              | Append of 'a * 'a
datatype ConsList = FoldC of ConsList C
datatype AppendList = FoldA of AppendList A
```

This version, which we will use for the rest of the examples in this note, keeps the definition of the type constructors C and A separate from the construction of their fixed points, which is identified by the explicit fold operations.

Given these recursively defined types, there are two natural ways to define functions based on their structure. If a function takes an object of a recursive type as an argument, then we may often cast it in the form of an *iteration*, or structural recursion, so that all we need to create is an auxiliary function which can do the work on one level of the recursive object. Dually, if a function *produces* an object of a recursive type as a result, then we may often cast it in the form of a *coiteration*, where we only need to specify how to generate the output one level at a time.

In more detail, suppose that we have a type τ satisfying the equation $\tau = C(\tau)$. We may define an iterative function from τ to another type σ by specifying an auxiliary function of type $C(\sigma) \to \sigma$. Given an argument of type τ, the process of iteration will first unfold one level of the argument, giving an object of type $C(\tau)$. It then visits all of the recursive components of type τ within this object (if any) and applies the iteration to each; in terms of a Divide-and-Conquer strategy, this corresponds to the (easy) splitting into subproblems and the recursive call on the function to solve each of them. At this point we have an object of type $C(\sigma)$, since each of the recursive call sites has been replaced with a result of type σ. Applying the auxiliary function then completes the outermost level of processing (the "hard join" part of the strategy), leaving the desired result of type σ.

A coiterative function from σ to τ may be defined analogously by providing an auxiliary function of type $\sigma \to C(\sigma)$. Given an argument of type σ, the process of coiteration first applies the auxiliary function to get an object of type $C(\sigma)$; the effect of this is to generate the outermost level of the result (the "hard split"), leaving subproblem objects of type σ in the appropriate components for the recursive call. After all of the subproblems are solved (mapping each from type σ to type τ), the resulting structure of type $C(\tau)$ may be folded back up into the final answer of type τ. Thus a characteristic of coiteration is that the auxiliary function is applied *before* the recursion, which matches the "hard split/easy join" version of Divide-and-Conquer. Examples of both of these styles of function are discussed in more detail below.

In terms of category theory, the function defined by iteration is the unique morphism from the initial C-algebra on τ (e.g., the function `FoldC` of type $C($`ConsList`$) \to$ `ConsList` defined above) to the (auxiliary) C-algebra of type $C(\sigma) \to \sigma$. Dually, the function defined by coiteration is the unique morphism from the (auxiliary) C-coalgebra of type $\sigma \to C(\sigma)$ to the final C-coalgebra on τ (in terms of our running example in ML, this would be the inverse function to `FoldC`, which is defined by `fun UnfoldC (FoldC x) = x`). When τ is (the carrier of) an initial C-algebra, it is in a sense a *least* fixed point of C, whereas a final C-coalgebra is a *greatest* fixed point. If a single type τ is to be both a least and a greatest fixed point of C, then we must also have that C represents an *algebraically compact* functor [2], but this is a technical matter that is easily ensured for the situations we will cover (for example, by assuming the existence of an evaluation function which will force elements of the final C-coalgebra type, which in general will include "infinite" elements, into the initial C-algebra type, perhaps at the cost of non-termination; see [14] for more details). This interpretation depends on being able to perform the standard extension of a parameterized type expression C into a functor by defining the meaning of $C(f)\colon C(\alpha) \to C(\beta)$ for an arbitrary function $f\colon \alpha \to \beta$.

For the example of Cons-lists, the parameterized type expression $C(\alpha) = 1 + ($`int` $\times \alpha)$ extends to a functor by defining C using the following ML code, which takes a function f of type $\alpha \to \beta$ and creates a function of type $C(\alpha) \to C(\beta)$ whose action is to apply f to any component of type α and leave the rest of the structure unchanged:

```
fun C f Nil                  = Nil
  | C f (Cons (head, tail)) = Cons (head, f tail)
```

Similarly, the functor A corresponding to Append-lists is given by

```
fun A f Empty                   = Empty
  | A f (Singleton n)           = Singleton n
  | A f (Append (left, right)) = Append (f left, f right)
```

Using these functors, we may define a function `IterC` which will take an auxiliary function f and produce the corresponding function which iterates f over Cons-lists:

```
fun IterC f x = f (C (IterC f) (UnfoldC x))
```

We may express this equivalently using pattern matching as

```
fun IterC f (FoldC x) = f (C (IterC f) x)
```

Conversely, here is a function `CoitC` which takes an auxiliary function g and produces its coiteration:

```
fun CoitC g x = FoldC (C (CoitC g) (g x))
```

The functions giving iteration and coiteration for Append-lists have the same form, with `FoldC` and `C` replaced by `FoldA` and `A`.

Since both the input and the output of a sorting function are lists, we may structure a sorting algorithm as either iteration or coiteration. When combined with the choice between Cons-lists and Append-lists, this leads to the four different algorithms under consideration. For example, given the following auxiliary function `insert` of type $1 + (\texttt{int} \times \texttt{ConsList}) \to \texttt{ConsList}$, we find that the function `(IterC insert)` implements Insertion Sort over Cons-lists:

```
fun isort l = IterC insert l

and insert Nil                  = FoldC Nil
  | insert (Cons (head, tail)) = insert1 head tail

and insert1 n (FoldC Nil) = FoldC (Cons (n, FoldC Nil))
  | insert1 n (FoldC (Cons (head, tail))) =
      if n<=head
      then FoldC (Cons (n, FoldC (Cons (head, tail))))
      else FoldC (Cons (head, insert1 n tail))
```

In words, `insert` handles two cases: if its argument is `Nil`, then it returns the empty list; otherwise, its argument is the `Cons` of an integer and a (sorted) Cons-list, so it calls `insert1` to insert the number into the appropriate position in the list. The presence of the explicit `FoldC` constructors makes this code somewhat uglier than necessary, but this is the trade-off for separating the construction of the recursive type from the definition of the functor C.

The remaining sorting functions are obtained by providing the appropriate auxiliary functions to select the least element of a list, merge two sorted lists, and partition a list around a pivot; Figs. 1, 2, 3, and 4 summarize the ML code necessary for implementing the four algorithms. The table given at the beginning may now be restated as follows:

	Cons-List	Append-List
Iteration	Insertion Sort	Merge Sort
Coiteration	Selection Sort	Quick Sort

3 Extending the type system

Consideration of the types of the objects to be manipulated has already guided the design of our sorting algorithms to the extent that we only need to come up with an auxiliary function of the appropriate type. We may obtain further guidance in the design of our algorithms if we consider an extension to the type system which will allow more properties of the data to be expressed. In particular, we will suppose the existence of a facility for constructing subtypes by specifying a characteristic function which determines membership in the subtype. We will use the syntax "`T st x => P x`" to describe the subtype of `T` selected by predicate $\texttt{P}: \texttt{T} \to \texttt{bool}$; the keyword `st` is chosen to suggest either "such that" or "subtype". As a simple example, the subrange type `Index = [1..10]` from Pascal may be described as

```
subtype Index = int st n => (1<=n) andalso (n<=10)
```

We may take this as a motivating example in that we propose that a naive implementation of the subtype system will use the same internal representation for elements of the subtype as for the supertype, with the addition of run-time checks to ensure that the predicate holds; a more sophisticated implementation should be able to avoid much of the run-time checking by performing an appropriate compile-time analysis. The precise details of this system are the subject of current research.

We would like the type of a sorting function to express that it takes arbitrary integer lists into the subtype of sorted lists. We may take advantage of the recursive definition of the list type and say that a list is sorted if the outermost level is sorted and also all the recursive instances are sorted. For the type of Cons-lists, this may be coded as

```
subtype 'a SC = 'a C st Nil                => true
                      | Cons (head, tail) => lower head tail
datatype SortConsList = FoldSC of SortConsList SC
```

where `lower` is the following function, which checks that its first argument is a lower bound of the elements in the second argument:

```
fun lower n (FoldC Nil)                  = true
  | lower n (FoldC (Cons (head, tail))) = (n<=head)
                                           andalso lower n tail
```

Using standard rules for reasoning about subtypes (see [1], for example), we may conclude that `SortConsList` is a subtype of `ConsList`.[1] We may now express the type of a sorting function on Cons-lists as `ConsList` → `SortConsList`. Since both the domain and the codomain are recursive types, we still have the choice between iteration and coiteration available. To define a sorting function by iteration, we now need to provide an auxiliary function of type C(`SortConsList`) → `SortConsList`; that is, the type of the function `insert` now makes it explicit that, in the non-`Nil` case, it will be given an integer and a *sorted* list, and must produce a sorted list as the result. Similarly, to define a sorting function by coiteration, the auxiliary function must have type `ConsList` → SC(`ConsList`); i.e., it takes an arbitrary Cons-list as input but produces either `Nil` or the `Cons` of an integer and a Cons-list *such that the integer is a lower bound of the list.* This is exactly the behavior we expect of the `select` function (strictly speaking, we also need to ensure that `insert` and `select` produce permutations of their inputs, but we omit this for simplicity).

Taking the same approach to augmenting the type construction for Append-lists, we may consider a predicate `lowers`: `AppendList` → `AppendList` → `bool` which holds if every element of the first list is a lower bound of the set of elements in the second list:

```
fun lowers (FoldA Empty) l = true
  | lowers (FoldA (Singleton n)) l = lowers1 n l
  | lowers (FoldA (Append (left, right))) l =
      lowers left l andalso lowers right l

and lowers1 n (FoldA Empty) = true
  | lowers1 n (FoldA (Singleton m)) = n<=m
  | lowers1 n (FoldA (Append (left, right))) =
      lowers1 n left andalso lowers1 n right
```

This allows us to specify the type of sorted Append-lists as a fixed point `SortAppendList` of the parameterized type expression SA given by

```
subtype 'a SA = 'a A st Empty => true
                      | Singleton n => true
                      | Append (left, right) =>
                          lowers left right
datatype SortAppendList = FoldSA of SortAppendList SA
```

Defining a sorting function for Append-lists then naturally involves either iteration of a function of type A(`SortAppendList`) → `SortAppendList` or coiteration of a function of type `AppendList` → SA(`AppendList`).

[1]The astute reader will note that this is not quite true, because we use a different fold constructor for `SortConsList`; we need to either provide the extra information that `FoldSC` corresponds to `FoldC`, or work in a more radical extension of ML which allows the fold operation to be implicit.

In the first case we find that the type of the `merge` function requires that (for non-trivial input) it take two sorted lists and produce a single sorted list as output, while in the second case we find that the `quick` function must split an arbitrary Append-list into two sublists satisfying the `lowers` predicate. Again, these augmented types provide very strong guidance for the construction of the necessary auxiliary functions.

4 Related work

Clark and Darlington [4] updated Darlington's original paper to obtain a cleaner derivation of the four sorting algorithms under consideration, not relying on the original's "generate-and-test" approach, nor on a specific coding of sequences as sets (that the derivations in [5] were not simple is supported by the presence of another follow-up paper in *Acta Informatica* [15], where Jacobs and Feather corrected two errors in Darlington's derivation of Quick Sort). They implicitly recognized the fourfold symmetry pointed out by Barstow in [3]. The derivations used by Barstow were presented in [8], which concentrated on building a knowledge base of techniques for programming. Other derivations which reference Darlington in motivating the choices that lead to the four algorithms include Smith [21, 22] and Traugott [24]. Smith's work is particularly interesting in that it analyzes the recursive phase of divide-and-conquer algorithms as the application of an appropriate algebra homomorphism, which necessarily preserves the surrounding structure – this is the essence of the categorical view discussed above, although Smith does not use any category theoretic terminology. All of these derivations are discussed, along with others, in the survey [23], which constructs composite derivations of the algorithms towards the goal of codifying programming knowledge, much in the spirit of Green and Barstow. Finally, as a more type-theoretic approach, Turner [25] extracts programs for Insertion Sort, Merge Sort, and Quick Sort as realizers of the corresponding correctness proofs, although he does not exploit the symmetry of the algorithms.

Other work on iteration and coiteration includes Hagino [10, 11], Mendler [19], Geuvers [7], Greiner [9], and Howard [12, 13, 14], all of which discuss typed languages (mostly extensions to the lambda calculus) with provisions for inductive and coinductive types and the natural operations over them. More general views of how to use such structures in programming, particularly in cases where computations may not terminate (which causes inductive and coinductive types to coincide), may be found in Kieburtz [16] and some of the papers from the Squiggol group [6, 18].

5 Acknowledgements

The ideas in this note were originally hashed out in discussion sessions with Dave Schmidt, John Hatcliff, and Jayant DeSouza at Kansas State University. The reference to the papers by Barstow and Merritt was provided by Richard Lorentz after a preliminary version of this paper was presented at California State University, Northridge. This work was partially supported by a grant from the Office of Naval Research.

References

[1] Amadio, R.M., Cardelli, L.: Subtyping recursive types. Technical Report 62, Digital Equipment Corporation, Systems Research Center, August 1990

[2] Barr, M.: Algebraically compact functors. J. Pure Appl. Algebra **82**, 211–231 (1992)

[3] Barstow, D.R.: Remarks on "A synthesis of several sorting algorithms" by John Darlington. Acta Inf. **13**, 225–227 (1980)

[4] Clark, K.L., Darlington, J.: Algorithm classification through synthesis. Comput. J. **23**(1), 61–65 (1980)

[5] Darlington, J.: A synthesis of several sorting algorithms. Acta Inf. **11**, 1–30 (1978)

[6] Fokkinga, M.M., Meijer, E.: Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, January 1991

[7] Geuvers, H.: Inductive and coinductive types with iteration and recursion. In: Nordström, B., Petersson, K., Plotkin, G. (eds.) Proceedings of the 1992 workshop on types for proofs and programs, June 1992

[8] Green, C.C., Barstow, D.R.: On program synthesis knowledge. Artif. Intell. **10**, 241–279 (1978)

[9] Greiner, J.: Programming with inductive and co-inductive types. Technical Report CMU-CS-92-109, Carnegie Mellon University, January 1992

[10] Hagino, T.: A categorical programming language. PhD thesis, University of Edinburgh, 1987

[11] Hagino, T.: A typed lambda calculus with categorical type constructors. In: Pitt, D.H, Poigne, A., Rydeheard, D.E. (eds.) Category theory and computer science (Lect. Notes Comput. Sci., vol. 283, pp. 140–157) Berlin Heidelberg New York: Springer 1987

[12] Howard, B.T.: Fixed points and extensionality in typed functional programming languages. PhD thesis, Stanford University, 1992

[13] Howard, B.T.: Inductive, projective, and retractive types. Technical Report MS-CIS-93-14, University of Pennsylvania, 1993

[14] Howard, B.T.: Fixpoint computations and coiteration. Technical Report KSU CIS 95-1, Kansas State University, 1995

[15] Jacobs, D., Feather, M.: Corrections to "A synthesis of several sorting algorithms" by J. Darlington. Acta Inf. **26**, 19–23 (1988)

[16] Kieburtz, R.B.: Inductive programming. Technical Report CS/E 93-001, Oregon Graduate Institute of Science and Technology, 1993

[17] Knuth, D.E.: The art of computer programming, vol. 3: sorting and searching. Reading, MA: Addison-Wesley 1973

[18] Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) Proceedings of the fifth ACM conference on functional programming languages and computer architecture (Lect. Notes Comput. Sci., vol. 523, pp. 124–144) Berlin New York: Springer 1991

[19] Mendler, N.P.: Recursive types and type constraints in second-order lambda calculus. In: Second annual IEEE symposium on logic in computer science, pp. 30–36, 1987

[20] Merritt, S.M.: An inverted taxonomy of sorting algorithms. Commun. ACM **28**(1), 96–99 (1985)

[21] Smith, D.R.: The design of divide and conquer algorithms. Sci. Comput. Program. **5**, 37–58 (1985)

[22] Smith, D.R.: Top-down synthesis of divide-and-conquer algorithms. Artif. Intell. **27**, 43–96 (1985) Reprinted in: Rich, C., Waters, R. (eds.) Readings in artificial intelligence and software engineering, pp. 35–61. Los Altos, CA: Morgan Kaufmann 1986

[23] Steier, D.M., Anderson, A.P.: Algorithm synthesis: a comparative study. New York: Springer 1989

[24] Traugott, J.: Deductive synthesis of several sorting algorithms. In: Siekmann, J.H. (ed.) Eighth international conference on automated deduction (Lect. Notes Comput. Sci., vol. 230, pp. 641–660) Berlin Heidelberg New York: Springer 1986

[25] Turner, R.: Constructive foundations for functional languages. London: McGraw-Hill 1991

A ML code for the four sorting algorithms

```
(* Recursive datatype of Cons-lists *)
datatype 'a C = Nil
              | Cons of int * 'a
datatype ConsList = FoldC of ConsList C

(* Action of C on functions, to make it a functor *)
fun C f Nil                 = Nil
  | C f (Cons (head, tail)) = Cons (head, f tail)

(* Iteration and Coiteration functionals for Cons-lists *)
fun IterC f (FoldC x) = f (C (IterC f) x)
fun CoitC g x = FoldC (C (CoitC g) (g x))


(* Insertion Sort by Iteration on Cons-lists *)
fun isort l = IterC insert l

(* Insert head element into (sorted) tail; do nothing if empty *)
and insert Nil                 = FoldC Nil
  | insert (Cons (head, tail)) = insert1 head tail

(* Insert n into a sorted list *)
and insert1 n (FoldC Nil) = FoldC (Cons (n, FoldC Nil))
  | insert1 n (FoldC (Cons (head, tail))) =
      if n<=head
      then FoldC (Cons (n, FoldC (Cons (head, tail))))
      else FoldC (Cons (head, insert1 n tail))
```

Figure 1: ML code for Insertion Sort

```
(* Selection Sort by Coiteration on Cons-lists *)
fun ssort l = CoitC select l

(* Select minimum element of list and put at head; do nothing if empty *)
and select (FoldC Nil) = Nil
  | select (FoldC (Cons (head, tail))) =
      case select tail of
          Nil              => Cons (head, tail)
        | Cons (min, rest) => if head<=min
                              then Cons (head, tail)
                              else Cons (min, FoldC (Cons (head, rest)))
```

Figure 2: ML code for Selection Sort

```
(* Recursive datatype of Append-lists *)
datatype 'a A = Empty
              | Singleton of int
              | Append of 'a * 'a
datatype AppendList = FoldA of AppendList A

(* Action of A on functions, to make it a functor *)
fun A f Empty                  = Empty
  | A f (Singleton n)          = Singleton n
  | A f (Append (left, right)) = Append (f left, f right)

(* Iteration and Coiteration functionals for Append-lists *)
fun IterA f (FoldA x) = f (A (IterA f) x)
fun CoitA g x = FoldA (A (CoitA g) (g x))


(* Functions to pick out first element/rest of elements from Append-list;
     raise EmptyList if no first element *)
exception EmptyList
fun first (FoldA Empty)                  = raise EmptyList
  | first (FoldA (Singleton n))          = n
  | first (FoldA (Append (left, right))) = (first left
                                            handle EmptyList => first right)
and rest (FoldA Empty)                  = raise EmptyList
  | rest (FoldA (Singleton n))          = FoldA Empty
  | rest (FoldA (Append (left, right))) = (FoldA (Append (rest left, right))
                                             handle EmptyList => rest right)


(* Merge Sort by Iteration on Append-lists *)
fun msort l = IterA merge l

(* Merge left and right (sorted) lists; do nothing if not an append *)
and merge Empty                  = FoldA Empty
  | merge (Singleton n)          = FoldA (Singleton n)
  | merge (Append (left, right)) = merge1 left right

(* Merge two sorted lists *)
and merge1 (FoldA Empty) l = l
  | merge1 (FoldA (Singleton n)) (FoldA Empty) = FoldA (Singleton n)
  | merge1 (FoldA (Singleton n)) (FoldA (Singleton m)) =
      if n<=m
      then FoldA (Append (FoldA (Singleton n), FoldA (Singleton m)))
      else FoldA (Append (FoldA (Singleton m), FoldA (Singleton n)))
  | merge1 (FoldA (Singleton n)) (FoldA (Append (left, right))) =
      if n<=(first right)
      then FoldA (Append (merge1 (FoldA (Singleton n)) left, right))
      else FoldA (Append (left, merge1 (FoldA (Singleton n)) right))
  | merge1 (FoldA (Append (left, right))) l = merge1 left (merge1 right l)
```

Figure 3: ML code for Merge Sort

```
(* Datatype used for return value of filter *)
datatype Partition = Null | Pair of AppendList * AppendList

(* Quick Sort by Coiteration on Append-lists *)
fun qsort l = CoitA quick l

(* Split list so elements in left half are smaller than those in right;
     do nothing if fewer than two elements *)
and quick (FoldA Empty) = Empty
  | quick (FoldA (Singleton n)) = Singleton n
  | quick (l as FoldA (Append (left, right))) =
      (let val pivot = first l in
         case filter pivot (rest l) of
             Null => Singleton pivot
           | Pair (low, high) =>
               Append (FoldA (Append (low, FoldA (Singleton pivot))), high)
       end
         handle EmptyList => Empty)

(* Partition list by comparing elements against the pivot *)
and filter pivot (FoldA Empty) = Null
  | filter pivot (FoldA (Singleton n)) =
      if pivot<=n
      then Pair (FoldA Empty, FoldA (Singleton n))
      else Pair (FoldA (Singleton n), FoldA Empty)
  | filter pivot (FoldA (Append (left, right))) =
      case filter pivot left of
          Null => filter pivot right
        | Pair (low, high) =>
            case filter pivot right of
                Null => Pair (low, high)
              | Pair (low1, high1) =>
                  Pair (FoldA (Append (low, low1)),
                        FoldA (Append (high, high1)))
```

Figure 4: ML code for Quick Sort